

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
D. Seitzer, B. Schmidt, H. J. Schneider, H. Wedekind

DETERMINISTISCHES ZUORDNEN VIELFACH DURCHLAUFENER AUFGABENSYSTEME IN MULTIPROZESSORSYSTEMEN

Beate Hoppe

DISSERTATION

BAND 17 · NUMMER 6 · ERLANGEN · OKTOBER 1984



ISSN 0344 - 3515

DETERMINISTISCHES ZUORDNEN VIELFACH DURCHLAUFENER AUFGABENSYSTEME IN MULTIPROZESSORSYSTEMEN

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt

von

Beate Hoppe

ERLANGEN 1984

Als Dissertation genehmigt von der Technischen
Fakultät der Universität Erlangen-Nürnberg

Tag der Einreichung:	20.06.1984
Tag der Promotion:	09.08.1984
Dekan:	Prof. Dipl.-Ing. G. Vetter
Berichterstatter:	Prof. Dr. F. Hofmann Prof. Dr. H.J. Schneider

Herr Professor Dr. F. Hofmann hat die Thematik dieser Arbeit mit großem Interesse aufgenommen. Für die Förderung und Betreuung dieser Dissertation gilt ihm mein besonderer Dank, ebenso Herrn Professor Dr. H.J. Schneider für die Übernahme des Zweitgutachtens.

Meinem Kollegen Dr. C.-U. Linster möchte ich für alle Anregungen und die sorgfältige Durchsicht dieser Arbeit danken.



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41546217_0035

Zusammenfassung

In dieser Arbeit wird die Problematik des Scheduling beliebiger deterministischer Aufgabensysteme auf Multiprozessorsysteme aufgegriffen. Es werden die Merkmale solcher Scheduling-Probleme und die daraus resultierenden Eigenschaften für einen Lösungsalgorithmus skizziert. Daran schließt sich die Schilderung verschiedener möglicher Lösungswege an, die sich optimaler, suboptimaler oder heuristischer Vorgehensweisen oder Kombinationen daraus bedienen.

Ziel der Arbeit ist es zu zeigen, wie leistungsfähig die Tauschverfahren insbesondere für den Problemkreis des Scheduling vielfach durchlaufener Aufgabensysteme sind. Die mittels eines dialoggesteuerten Experimentiersystems zum Testen unterschiedlicher Kombinationen von Verfahren erzielten Ergebnisse beim Vergleich von verschiedenen Varianten der Tauschverfahren werden erörtert.

Abschließend wird über den Einsatz eines auf den gewonnenen Erfahrungen basierenden, heuristischen, vollautomatischen Verfahrens in einem konkreten Anwendungsfall, dem Projekt MUPSI, berichtet.

Abstract

The problem of scheduling any deterministic task system on a multiprocessor-system is picked up. The characteristics of such scheduling-problems and the therefrom resulting attributes of an algorithm to there solving are outlined. Different manners for solving are described, which make use of optimal, suboptimal or heuristic procedures or combinations of it.

The paper demonstrates the efficiency of the interchange methods particularly for the problems of scheduling task systems, which have to be performed periodically. The results of tests of diverse combinations of procedures to compare different variants of the interchange methods are discussed, which are obtained by the means of an experimental system, which is controlled in a dialog.

Finally the use of a heuristic, fully automatic method, which is based on the gained experiences, in a concrete application, the project MUPSI, is reported.

INHALTSVERZEICHNIS

	Seite
1. Einleitung	3
1.1. Motivation und Zielsetzung	3
1.2. Übersicht	8
2. Problemdefinition	10
2.1. Grundbegriffe	10
2.2. Klassifikation des Problems	12
2.3. Anforderungen an einen Algorithmus	16
2.4. Die Klassen P und NP	19
3. Lösungsverfahren	26
3.1. Optimale Verfahren	28
3.1.1. Lineare Programmierung	28
3.1.1.1. Allgemeine LP-Verfahren	29
3.1.1.1.1. Schnittebenen- Verfahren	30
3.1.1.1.2. Entscheidungsbaum- Verfahren	31
3.1.1.2. Integer-Programmierung	31
3.1.1.2.1. Allgemeines	31
3.1.1.2.2. 0/1-Verfahren	32
3.1.1.3. Exkurs über NLP-Verfahren	37
3.1.2. Enumeration	38
3.1.2.1. "Branch and Exclude"	41
3.1.2.2. "Branch and Bound"	42
3.1.2.2.1. Grundlegende Prinzipien	42
3.1.2.2.2. Einige Verfahren	45
3.1.2.3. Dynamische Programmierung	50
3.1.3. Graphentheoretische Verfahren	53
3.2. Suboptimale Verfahren	63
3.2.1. Greedy-Methoden	67
3.2.2. Exakte Verfahren mit heuristischem Abbruchkriterium	70

	Seite
3.2.3. Rein heuristische Verfahren	72
3.2.3.1. Graphentheoretische und Clustering-Verfahren	73
3.2.3.2. Aufspaltung des Problems	77
4. Nachbarschaftssuchverfahren	79
4.1. Grundlegende Prinzipien	79
4.1.1. Die Ausgangslösung	80
4.1.2. Die Nachbarschaft	81
4.1.3. Auswahl- und Entscheidungskriterien	84
4.2. Einige Verfahren	86
4.2.1. Direkte Suchverfahren	89
4.2.2. Tauschverfahren	93
4.2.3. Zusammenfassung	95
5. Vergleich von Tauschverfahren bei Scheduling- Problemen	96
5.1. Motivation	96
5.2. Vergleichstests	100
5.2.1. Voraussetzungen	100
5.2.2. Ergebnisse	106
6. Das Zuordnungsverfahren bei MUPSI	114
6.1. Anforderungen	114
6.2. Lösungsweg	118
6.2.1. Finden einer Erstlösung	118
6.2.2. Verbesserung der Ausgangslösung	119
6.2.3. Teilung von Blöcken	121
7. Schlußbemerkungen	124
Anhang	126
Literaturverzeichnis	141

1. Einleitung

1.1. Motivation und Zielsetzung

Je mehr sich der Mensch die Fähigkeiten und möglichen Einsatzgebiete einer Rechenanlage erschließt, desto größer ist sein Wunsch nach noch leistungsfähigeren Rechnern. Die gewünschte Leistungssteigerung wird in zunehmendem Maße nicht nur durch noch komplexere und schnellere Großrechenanlagen realisiert, sondern vermehrt auch durch geeignete Zusammenschaltung mehrerer, oft kleiner Rechenanlagen zu Multiprozessorsystemen.

Die Möglichkeiten des Einsatzes und der Realisierung solcher Systeme sind sowohl in der Industrie, als auch an den Universitäten ein Gebiet intensiver Untersuchungen. Über den Aufbau und Betrieb verschiedenartigster Multiprozessorsysteme, ihre Möglichkeiten und die auftretenden Probleme liegen schon manche Berichte vor. Jedoch wird ein Aspekt bisher meist völlig unberücksichtigt gelassen, der doch einen nicht geringen Einfluß auf den effektiven Einsatz einer solchen komplexen Rechnerstruktur hat.

Bei Multiprozessor-Projekten werden z. Z. viele Überlegungen angestellt, wie eine effiziente Hardware für ein solches System von Rechnern auszusehen hat. Kaum einer macht sich jedoch Gedanken darüber, wie diese effiziente Hardware auch von den künftigen Anwendern effektiv genutzt werden kann, d. h. wie ihnen bei den Problemen geholfen werden kann, die mit dem Einsatz derartiger neuen Systeme verstärkt auftreten. Denn den Vorteilen dieser komplexeren Rechnerstrukturen, wie z. B. die Steigerung der Verarbeitungsgeschwindigkeit durch Parallelarbeit mehrerer Rechner, stehen auch größere Schwierigkeiten in der Organisation gegenüber, sei es beispielsweise in der Speicherverwaltung, der Betriebsmittelzuteilung, der Koordination der Prozesse oder der Prozessorvergabe. Hier soll nun eines dieser Probleme herausgegriffen und ein Lösungsweg aufgezeigt werden.

Der Anwender muß sich bisher - will er diese neue Hardware nutzen - erst mit den speziellen Möglichkeiten des ihm vorliegenden Mehrrechnersystems vertraut machen. Dabei stellt sich ihm besonders ein Problem, das bei den bisher von ihm benutzten Monorechnern nicht auftrat: Die verschiedenen Aufgaben, die er erledigt zu haben wünscht, müssen geeignet auf die einzelnen Prozessoren des Multiprozessorsystems verteilt werden. Zur Lösung dieses Problems werden ihm meist keine Hilfsmittel zur Verfügung gestellt, so daß er seine Programme "von Hand" den einzelnen Rechnern zuordnen muß.

Einfache Aufgabensysteme lassen sich sicher auch auf diese Weise noch einigermaßen gut verteilen. Bei komplexen Programmsystemen stößt dies aber sehr schnell an die Grenzen des vom menschlichen Gehirn überschaubaren. Je komplexer daher ein solches Aufgabenpaket ist, desto größer wird zwangsweise die Abweichung von einer effizienten Verteilung sein. Somit geht dem Anwender in zunehmendem Maße die Leistungssteigerung, die er mit Hilfe des Multiprozessorsystems zu erreichen hoffte, trotz der effizienten Hardware wieder verloren.

In dieser Arbeit soll nun ein Verfahren vorgestellt werden, das ein vom Anwender spezifiziertes Aufgabensystem auf die verfügbare Hardware eines Multiprozessorsystems verteilt. Dieses Verfahren arbeitet vollautomatisch. Dadurch wird der Anwender, der ja nicht immer aus dem Bereich der Informatik kommt, davon befreit, sich bei der Benutzung eines Multiprozessorsystems erst in die speziellen Besonderheiten des ihm gerade vorliegenden Systems einarbeiten zu müssen, um eine möglichst optimale Verteilung seiner Programme zu erreichen. Für ihn soll der Unterschied zu der Verwendung eines einzelnen Rechners nur in der Leistungssteigerung liegen.

Zur Verfügung stehen m Rechner, auf die n Programme verteilt werden sollen, wobei eventuell irgendwelche Restriktionen berücksichtigt werden müssen (z. B. auf Hardware-Seite Speicherplatzbeschränkungen oder auf Software-Seite Abhängigkeiten der Programme untereinander). Was für Verfahren sind nun für diese Problematik zur Lösung in Betracht zu ziehen?

Es gibt eine Vielzahl von Scheduling-Problemen, die jeweils verschiedene Lösungsansätze bedingen. Denn wie in Kapitel 2 noch näher erläutert, wird es nie ein allgemeines Verfahren geben, mit dem alle Probleme einer Klasse (hier des Scheduling) gelöst werden können. Die Klasse der Scheduling-Aufgaben läßt sich nun in große Gruppen gliedern aufgrund der Art der vorliegenden Angaben über die Programme und Rechner:

einerseits

- die Gruppe der stochastischen und
- die der deterministischen Schedules,

andererseits

- die Gruppe der einmal durchlaufenen und
- die der vielfach (periodisch) durchlaufenen Programme.

Dies sei durch zwei Beispiele illustriert:

Will man z. B. durch solch ein Multiprozessorsystem die Leistung eines Einzelrechners, an dem mehrere Benutzer gleichzeitig arbeiten, erhöhen, handelt es sich um eine Art Rechenzentrumsbetrieb. Charakteristisch für solche Systeme ist es, daß laufend ein Wechsel im Prozeßsystem zu verzeichnen ist, d. h. daß Benutzer sich an- und abmelden, neue Prozesse hinzukommen oder fertiggestellte das System verlassen. Dabei ist natürlicherweise nicht vorhersehbar bzw. im voraus bekannt, wann solch ein Wechsel im Prozeßgefüge stattfindet. Dies erfordert Scheduling-Algorithmen, die auf stochastischen Daten über die Prozesse arbeiten. Außerdem braucht man in diesem Fall schnelle Algorithmen, da wegen des ständigen Wechsels diese Routinen sehr häufig durch-

laufen werden müssen. Schnellere Verteilungsverfahren sind hier solchen vorzuziehen, die bessere Lösungen liefern, da bei dieser Art von Verteilungsproblemen selten das absolute Optimum zwingend notwendig ist. Man kann sich meist mit einer guten Approximation der optimalen Lösung zufrieden geben.

Im Gegensatz dazu gibt es eine ganze Reihe von Systemen, in denen keine auf Wahrscheinlichkeiten beruhende Daten verwendet werden müssen. Man denke nur an den immer stärker werdenden Einsatz von Industrierobotern in Fertigungsstraßen. Bei der Steuerung derartiger Systeme mit Echtzeitanforderungen ist jeglicher Bedarf an Rechenzeit, Speicher usw. meist im voraus bekannt. Es muß bei dieser Anwendungsklasse also ein von vornherein bekanntes, sich nicht (oder nur selten) durch hinzukommende oder wegfallende Programme änderndes Aufgabensystem verteilt werden. Da es sich bei solchen Anwendungen im allgemeinen um Programme handelt, die periodisch immer wieder abgearbeitet werden, bleibt die einmal vorgenommene Zuordnung während des gesamten Betriebs erhalten, d. h. es handelt sich sozusagen um residente Programme. Hier liegt die Hauptanforderung an den Zuordnungsalgorithmus in erster Linie in einer möglichst optimalen Verteilung der Programme (wobei sich das Optimierungskriterium natürlich wieder an dem gewählten Einsatzgebiet orientiert). Eine untergeordnete Rolle spielt die Zeit, die der Verteilungsalgorithmus selbst braucht, da er in diesem Fall ja nur selten, meist nur einmal am Anfang einer Anwendung, läuft.

Als Beispiel für die zuletzt geschilderte Scheduling-Problematik, die des deterministischen Scheduling vielfach durchlaufener Programme, sei das Projekt MUPSI/Bol83a/ erwähnt, bei dem ein Multiprozessorsystem zur Echtzeitsimulation nachrichtentechnischer Systeme eingesetzt wird. Das Aufgabensystem setzt sich aus einer Reihe von Programmen (Blöcken) zusammen, die jeweils Teilaufgaben der gesamten Simulation übernehmen (z. B. Filterung, FFT, Modulation, u. ä.). Alle Programme des Aufgabensystems werden einmal vor

Beginn der Simulation auf die Prozessoren des Systems verteilt und dann immer wieder in einem bestimmten Zeitraster periodisch durchlaufen. Bei jedem Durchlauf wird der nächste Wert einer Datenquelle (z. B. der gerade abgetastete Wert einer realen Komponente) als Eingabe verarbeitet. Die Gesamtgeschwindigkeit des Systems (das Zeitraster, die Abtastrate) wird hier durch den am stärksten belasteten Prozessor bestimmt. Deshalb wird eine möglichst gleichmäßige Auslastung der beteiligten Prozessoren angestrebt.

Das hier angeführte vielfach durchlaufene Aufgabensystem ist keine Besonderheit des geschilderten Projekts. Bei vielen sich der Prozeßsteuerung und -überwachung stellenden Aufgaben kommen solche sich periodisch wiederholende Programmabläufe vor. Wie schon gezeigt, müssen bei solchen Anwendungen ganz andere Anforderungen an ein Verteilungsverfahren gestellt werden als bei einem Scheduling von Programmen in einem Mehrbenutzersystem; hier nämlich Trachten nach Lösungen, die möglichst optimal sind, im Gegensatz zu möglichst schnellen Algorithmen bei Scheduling-Problemen der zuerst geschilderten Art.

Ziel dieser Arbeit ist es nun zu zeigen, wie leistungsfähig Tauschverfahren für den geschilderten Problemkreis des deterministischen Scheduling bei periodischen Aufgabensystemen sind. Die Tauschverfahren, eine Klasse heuristischer, suboptimaler Algorithmen, sind im Bereich des Operations Research seit längerem bekannt, jedoch liegen bisher keinerlei experimentelle Untersuchungen über die Eigenschaften der verschiedenen Varianten vor. Für das Anwendungsgebiet der Zuordnung eines Aufgabensystems auf ein Multiprozessorsystem sind sie unverständlicherweise sogar bisher völlig unbeachtet geblieben.

Die Tauschverfahren selbst sind nicht auf diesen engen Problemkreis beschränkt. Prinzipiell sind sie z. B. auch für andere Schedulingprobleme einsetzbar. Die Mächtigkeit

dieser Verfahren zeigt sich indessen überwiegend bei komplexeren Problemstellungen. Bei einfachen Zuordnungsaufgaben sind ihnen andere, eventuell sogar optimale Verfahren manchmal überlegen. Insbesondere lassen sich für Probleme, bei denen es auf einen schnellen Algorithmus ankommt, oft geeignetere Verfahren finden.

1.2. Übersicht

Nach diesem ersten einführenden Kapitel wird im nächsten näher auf die Problematik einer automatischen Zuordnung des eingangs geschilderten Anwendungsgebiets, die speziellen Anforderungen an einen Algorithmus dafür und die daraus resultierenden Eigenschaften eingegangen.

Kapitel 3 enthält einen allgemeinen Überblick über gebräuchliche Verfahren zur Lösung der Zuordnungsproblematik, wobei insbesondere auch die Vor- und Nachteile dieser Verfahren für die in Kapitel 2 skizzierte Aufgabenstellung dargelegt werden.

Eine detaillierte Darstellung der Tauschverfahren mit ihren Varianten und den sich damit ergebenden Möglichkeiten und Problemen erfolgt anschließend in Kapitel 4.

Auf die Problematik des Scheduling vielfach durchlaufener Aufgabensysteme wird dann speziell im fünften Kapitel eingegangen. Es werden einige Ergebnisse von Vergleichstests der in Kapitel 4 beschriebenen Varianten bei den Tauschverfahren angegeben und die daraus zu ziehenden Schlußfolgerungen aufgezeigt.

Als Anwendungsbeispiel wird noch einmal auf das schon erwähnte Projekt MUPSI eingegangen. Es werden die durch die Anwendung gestellten Anforderungen an den Verteilungsalgorithmus beschrieben. Daraus resultierte ein Zuordnungsverfahren, das sich aus einer Kombination verschiedener in den Kapiteln 3 und 4 geschilderter Algorithmen zusammen-

setzt.

Abschließend werden die erzielten Ergebnisse noch einmal zusammengefaßt und ein Ausblick auf weitere Forschungsmöglichkeiten gegeben.

2. Problemdefinition

2.1. Grundbegriffe

Bevor näher beschrieben wird, welche Scheduling-Probleme in dieser Arbeit betrachtet werden, sollen erst einmal einige Grundbegriffe festgelegt werden.

- * Ein Scheduling-Problem stellt sich meist in Form eines Aufgaben-Systems.
- * Ein Aufgaben-System setzt sich aus Aktivitätseinheiten zusammen, den Aufgaben.

(Im folgenden werden die Begriffe Aufgaben-System, Programm-System und Prozeß-System synonym verwendet, entsprechend auch die Begriffe Aufgabe, Prozeß und Programm.)

Man könnte ein Scheduling-Problem auch auf einer tieferen Ebene betrachten, indem man einen Ablaufplan für ein Programm, bestehend aus verschiedenen Operationen sucht. Solch eine Operation kann je nach Anwendung z. B. eine Prozedur, eine Instruktion oder eine Mikroinstruktion sein oder irgend etwas anderes, was als Teil eines größeren Ganzen betrachtet werden kann. Im Rahmen dieser Arbeit spielt die genaue Natur einer solchen unteilbaren Aktivitätseinheit keine Rolle. Wichtig ist nur ihr äußeres Verhalten, wie z. B. ihre Betriebsmittelanforderungen, ihre Bearbeitungszeit und ihre Beziehung zu anderen Operationen. Ohne Beschränkung der Allgemeinheit verstehe man daher im folgenden unter Scheduling-Problemen das Scheduling eines Aufgabensystems.

- * Die Bearbeitungszeit einer Aufgabe sei der Zeitaufwand, der zu ihrer Bearbeitung erforderlich ist.

* Ankunftszeiten legen für jede Aufgabe den frühest möglichen Zeitpunkt fest, an dem mit ihrer Bearbeitung begonnen werden kann. Sie können explizit benannt werden, falls eine Aufgabe Informationen braucht, die erst zu einem bestimmten Zeitpunkt vorliegen. Sie können aber auch implizit durch Abhängigkeiten in der Bearbeitungsreihenfolge der Aufgaben, den sogenannten Vorgängerrelationen, gegeben sein.

* Aufgaben können zu ihrem Ablauf bestimmte Betriebsmittel benötigen. Das sind etwa Speicherplatz, periphere Geräte, Daten usw., aber auch der Prozessor selbst kann als ein spezielles Betriebsmittel betrachtet werden.

Betriebsmittel lassen sich in verschiedener Weise klassifizieren:

- nach ihrem Hardware- und Softwarecharakter;
- nach ihrer Wiederverwendbarkeit (Speicher, Prozessoren, periphere Geräte stehen üblicherweise nach ihrer Verwendung erneut zur Verfügung, im Gegensatz zu Daten);
- nach ihrer Unterbrechbarkeit (Hauptspeicher und Prozessoren können entzogen werden);
- nach ihrer Teilbarkeit (Prozessoren und Geräte sind unteilbar, d. h. sie können immer nur einer Operation zur Zeit zugeordnet sein; Speichermedien sind dagegen im allgemeinen teilbar).

* Ein Schedule für ein Aufgaben-System ist die Festlegung der Bearbeitungsreihenfolge der Aufgaben auf den Prozessoren, unter Berücksichtigung der Bearbeitungszeiten und Betriebsmittelanforderungen der einzelnen Aufgaben. Zu einem Problem gibt es meist mehrere Schedules, unter denen dann aufgrund einer vorgegebenen Bewertungs-

funktion ein bestmöglicher Schedule ausgewählt wird. Solche Optimierungskriterien können z. B. sein:

- Das gesamte Aufgaben-System soll in minimaler Zeit ausgeführt werden. Das bedeutet Maximierung des Durchsatzes des Systems.
 - Die mittlere Wartezeit der Aufgaben soll möglichst minimal sein.
 - Für jede Aufgabe ist ein Fertigstellungstermin vorgegeben, dessen Überschreitung mit einer Verzugsstrafe belegt wird. Es wird angestrebt, die Summe der Verzugsstrafen aller Aufgaben des Systems minimal zu halten.
 - Bei vorgegebenen Fertigstellungsterminen soll die Anzahl der terminüberschreitenden Aufgaben minimiert werden.
- usw.

2.2. Klassifikation des Problems

Jedes Scheduling-Problem wird durch verschiedene Größen charakterisiert:

- die Anzahl der Aufgaben
- die Anzahl der Prozessoren
- die Reihenfolge, in der die Prozessoren einer Aufgabe zugeteilt werden müssen
- die Bearbeitungszeiten der Aufgaben

- Ankunfts- und Fertigstellungszeiten der Aufgaben
- Kosten
- usw.

Üblicherweise wird ein Scheduling-Problem durch vier Parameter beschrieben:

(A, B, C, D)

- A Angaben über die Aufgaben, wie Anzahl, Ankunftszeiten, Reihenfolgebedingungen, usw.
 - B Angaben über die Prozessoren (allgemeiner auch Maschinen), wie Anzahl, Geschwindigkeiten, verfügbare Betriebsmittel, usw.
 - C Angaben über den Fluß der Aufgaben durch das Prozessor-System
- = F Flow-Shop: Alle Aufgaben durchlaufen alle Prozessoren in der selben Reihenfolge.
 - = P permutierender Flow-Shop: Nicht nur die Reihenfolge der Prozessoren ist für alle Aufgaben dieselbe, sondern auch die Reihenfolge der Prozesse auf einer Anlage. Dadurch ist ein Schedule durch eine Permutationsnummer völlig beschrieben.
 - = G allgemeines Job-Shop-Problem: Es gibt keinerlei Einschränkungen technologischer Art oder in der Reihenfolge und Bearbeitungszeit der einzelnen Aufgaben.
 - = I Parallel-Shop: Jede Aufgabe besteht aus einer Operation (Aktivitätseinheit); die Reihenfolge der Prozessoren liegt nicht fest.

D gibt die Bewertungsfunktion, das Optimierungskriterium an (wie z. B. maximaler Durchsatz, minimale Wartezeit usw.).

Zur Veranschaulichung: Ein System mit n Aufgaben und m Prozessoren sei gegeben. Es sollen keinerlei Beschränkungen vorliegen, und die Bearbeitungszeit des gesamten Aufgaben-Systems soll möglichst minimal sein. In obiger Beschreibungsform eines Scheduling-Problems hätte man es also in diesem Fall mit einem Problem der Art

$$(n, m, G, C_{\max})$$

zu tun, wobei $C_{\max}$ bedeutet, daß das Maximum der Bearbeitungszeiten (C_i) auf einem Prozessor minimiert werden soll.

Als nächstes sollen einige Annahmen folgen, die allen weiteren Überlegungen zugrundeliegen sollen, und auf diese Weise die hier betrachteten Probleme in der großen Klasse der Scheduling-Probleme noch mehr abgegrenzt werden.

1. Jede Aufgabe ist eine Einheit.

D. h. obwohl sich ein Aufgabe aus mehreren Operationen zusammensetzt, können nicht zwei dieser Operationen gleichzeitig bearbeitet werden.

2. Keine Unterbrechungen.

Jede einmal gestartete Aufgabe muß erst beendet werden, bevor auf demselben Rechner ein anderes Programm gestartet werden kann.

3. Kein Abbruch.

Jede Aufgabe muß bis zur Beendigung bearbeitet werden.

4. Die Bearbeitungszeiten sind unabhängig vom Schedule.

5. Es kann vorkommen, daß ein Rechner zu einem Zeitpunkt keine Aufgabe bearbeitet, d. h. daß er "idle" ist.
6. Kein Rechner kann mehr als eine Aufgabe zur Zeit bearbeiten.
7. Rechner fallen nie aus und sind für die Dauer der Scheduling-Periode verfügbar.
8. Die technologischen Beschränkungen sind im voraus bekannt und unveränderlich.
9. Es gibt keine Zufälligkeiten.
Insbesondere gilt:
 - a) Die Anzahl der Aufgaben ist bekannt und unveränderlich.
 - b) Die Anzahl der Rechner ist bekannt und unveränderlich.
 - c) Die Bearbeitungszeiten sind bekannt und unveränderlich.
 - d) Die Ankunftszeiten der Programme sind bekannt und unveränderlich.
 - e) Alle anderen, ein vorliegendes Problem bestimmenden Parameter sind bekannt und unveränderlich.

Diese Annahmen sind ohne Zweifel restriktiv. Aber besonders die Annahmen 8 und 9 unterscheiden sich von den anderen. Sie beschränken die Klasse der im folgenden betrachteten Scheduling-Probleme auf solche, in denen im voraus alle zahlenmäßigen Anforderungen bekannt und gleichbleibend sind. Es gibt keine Unsicherheit.

Def.: Probleme, bei denen die Ankunftszeiten der Aufgaben bekannt und unveränderlich sind, nennt man statisch und solche, in denen die Prozesse zu unvorhersehbaren Zeiten eintreffen, dynamisch.

Def.: Probleme, in denen alle Anforderungen aller Aufgaben im voraus bekannt und unveränderlich sind, heißen deterministisch und Probleme, bei denen der Bedarf an Rechenzeit usw. nicht abschätzbar ist, stochastisch.

Man kann davon ausgehen, daß die meisten in der Praxis vorkommenden Scheduling-Probleme dynamisch und stochastisch sind (siehe auch /Soc83a/). In diese Gruppe gehört auch das im vorigen Kapitel geschilderte Scheduling-Problem, das bei einem Rechenzentrumsbetrieb auftritt. Aber, wie ebenfalls in dem einleitenden Kapitel motiviert wurde, gibt es durchaus eine Reihe aus der Praxis entstammender wichtiger Aufgabenstellungen, für die die eben getroffenen Annahmen zutreffen. Aufgrund obiger Definitionen können diese Scheduling-Probleme, die in dieser Arbeit im weiteren betrachtet werden sollen, als statisch und deterministisch bezeichnet werden.

2.3. Anforderungen an einen Algorithmus

Nun ist ein Algorithmus gesucht, um für Zuordnungsprobleme der gerade festgelegten Art die beste Zuordnung bezüglich einer bestimmten Zielvorstellung auszuwählen. Die einfachste Methode ist es, alle erlaubten Reihenfolgen zu generieren (es gibt ja nur endlich viele Möglichkeiten), ihnen einen Zielfunktionswert zuzuordnen und sich daraus den besten Schedule auszuwählen. Dieses Enumerationsverfahren ist jedoch praktisch nicht durchführbar. Es zeigt sich, daß die Zeit, die dieser Algorithmus braucht, mit steigender Anzahl von Aufgaben und Prozessoren sehr schnell wächst, üblicherweise exponentiell mit der Problemgröße. Solch ein Algorithmus kann nicht als ein guter bezeichnet werden. Aber gibt es bessere?

Um die Brauchbarkeit von Algorithmen vergleichen zu können, gibt es verschiedene Kriterien. Die wichtigsten dieser Art sind wohl der Bedarf der Verfahren an Speicherplatz und vor allem Bearbeitungszeit in Abhängigkeit von einer das Problem charakterisierenden Größe. Solch eine Problemgröße kann z. B. der Grad eines Polynoms bei Polynom-Problemen oder die Zahl der Kanten bei Graphen-Problemen sein. Bei Scheduling-Problemen, wie z. B. die in dieser Arbeit behandelten, könnte man die Zahl der Aufgaben und Prozessoren als Maß festlegen.

Def.: Die Anzahl der Schritte, die ein Algorithmus zum Lösen eines Problems abhängig von seiner Größe braucht, heißt Zeitkomplexität. Analog wird die Speicherkomplexität definiert als der zum Lösen der Aufgabe benötigte Speicherbedarf in Abhängigkeit von der Problemgröße.

In erster Linie ist jedoch die Zeitkomplexität eines Algorithmusses von Interesse. Mit ihrer Hilfe kann man Probleme in solche klassifizieren, die praktisch berechenbar sind, und in solche, die es nicht sind, deren Lösung also nur mit unvertretbar hohem Zeitaufwand möglich ist. Die Zeitkomplexitätsfunktion $f(n)$ eines Algorithmusses gibt die Zeitkomplexität an, die das Programm zum Lösen eines Problems der Größe n braucht. Es ist in der Praxis nicht nötig, diese Funktion vollständig zu berechnen. Was man braucht, ist ja nur eine Angabe darüber, wie das Verhalten des Algorithmusses bei wachsender Problemgröße ist.

Def.: Die Zeitkomplexitätsfunktion $f(n)$ ist von der Ordnung $O(g(n))$, d. h. f und g sind von derselben Ordnung, wenn

$$f(n) / g(n) \longrightarrow \text{konstant, wenn } n \longrightarrow \infty$$

Dabei gelten folgende Eigenschaften: Wenn die Komplexität von $f(n)$ von der Ordnung $O(n^a)$ ist und die von $g(n)$ von der Ordnung $O(n^b)$, dann gilt:

1. $f(n) \cdot g(n)$ ist von der Ordnung $O(n^{a+b})$;
2. $f(n)/g(n)$ ist von der Ordnung $O(n^{a-b})$;
3. $f(g(n))$ ist von der Ordnung $O(n^{ab})$.

Def.: Ein Algorithmus hat polynomiale Zeitkomplexität, wenn seine Zeitkomplexitätsfunktion $f(n)$ von der Ordnung $O(p(n))$ für ein Polynom $p(n)$ ist. Sonst hat er nichtpolynomiale Zeitkomplexität, typischerweise exponentielle.

Nun ist der Zeitbedarf eines polynomial begrenzten Algorithmusses bei wachsender Problemgröße wesentlich weniger dramatisch als bei andern. Bei Algorithmen mit nichtpolynomialer Zeitkomplexität steigen die Ausführungszeiten bei wachsender Problemgröße so schnell an, daß schon Probleme mittlerer Größe nicht mehr praktisch lösbar sind. Außerdem hat sich gezeigt, daß die üblicherweise verwendeten polynomial begrenzten Algorithmen nur von der Ordnung $O(n^2)$ oder $O(n^3)$ sind. So ist der Unterschied zwischen diesen beiden Klassen von Algorithmen sehr markant.

Das schnelle Ansteigen des Zeitbedarfs bei nichtpolynomial begrenzten Algorithmen hat überdies noch eine weitere unangenehme Auswirkung. Bei steigender Computerleistung kann man bei Algorithmen mit polynomialer Zeitkomplexität eine entsprechend multiplikativ wachsende Problemgröße lösen, wohingegen bei einem Algorithmus mit nichtpolynomialer Zeitkomplexität nur ein additiver Zuwachs möglich ist. D. h. wenn sich für das vorliegende Problem ein polynomial begrenzter Algorithmus finden läßt, weiß man, daß man bei Fortschritten in der Computer-Technologie viel umfang-

reichere Probleme wird lösen können. Hat man dagegen nur einen nichtpolynomial begrenzten Algorithmus gefunden, werden einem selbst solche Fortschritte nur wenig nützen. Sehr gut veranschaulichen die Tabellen 1.2. und 1.3. in dem Buch von Garey und Johnson/Gar79a/ (Seite 7-8) diese Eigenschaft.

Ein Algorithmus soll also künftig dann gut genannt werden, wenn er eine polynomiale Zeitkomplexität hat.

2.4. Die Klassen P und NP

Im letzten Abschnitt wurden zwei Arten von Algorithmen unterschieden, die mit polynomialer Zeitkomplexität und die mit nichtpolynomialer Zeitkomplexität. Mit ihrer Hilfe können nun wiederum auch die Probleme in zwei Klassen eingeteilt werden.

Def.: Die Klasse P enthält alle Probleme, für die Algorithmen mit polynomialem Zeitverhalten gefunden wurden. Die Klasse NP umfaßt alle Probleme, für die Algorithmen mit nichtpolynomialer Zeitkomplexität gefunden wurden.

Es gibt auch noch andere Möglichkeiten, diese Klassen und die noch folgenden Begriffe zu definieren, z. B. mit Hilfe des Konzepts der Turing-Maschine/Kar72a/ oder deterministischer und nichtdeterministischer Algorithmen/Hor81a/. Alle diese Beschreibungen sind jedoch äquivalent, so daß es gleich bleibt, welche man verwendet.

Offensichtlich ist P in NP enthalten, denn für jedes Problem mit polynomial begrenztem Algorithmus kann man leicht einen mit nichtpolynomialem Verhalten finden. Nun hat sich aber gezeigt, daß es Probleme gibt, für die man nie

einen polynomial begrenzten Algorithmus finden wird. Läßt sich also für die Klasse der in dieser Arbeit behandelten Probleme ein solcher finden?

Üblicherweise enthalten die Klasse P und NP nur Entscheidungsprobleme, nicht aber Optimierungsprobleme, zu denen auch die Lösung von Scheduling-Problemen gehört. Entscheidungsprobleme zeichnen sich dadurch aus, daß sie als Antwort nur "ja" oder "nein" erwarten. Jedoch ist diese Einschränkung nicht weiter einschneidend. Jedem Optimierungsproblem kann ein Entscheidungsproblem zugeordnet werden, bei dem entschieden werden soll, ob ein bestimmter Wert erreicht, unter- oder überschritten wurde. Ein Algorithmus, der diese Aufgabe löst, muß als erstes das Optimierungsproblem lösen und kann dann z. B. entscheiden, ob das ermittelte Optimum unterhalb einer bestimmten Schranke liegt. Dieser Algorithmus wird also nicht wesentlich mehr Zeit brauchen, als der Algorithmus zur Lösung des Optimierungsproblems. Wenn also das diesem zugeordnete Entscheidungsproblem in polynomialer Zeit beantwortet werden kann, ist auch das Optimierungsproblem in polynomialer Zeit lösbar. D. h. das Optimierungsproblem und das zugehörige Entscheidungsproblem sind gleich schwer zu lösen.

Wie kann man nun feststellen, zu welcher Klasse ein bestimmtes Scheduling-Problem gehört? Dies ist eine sehr wichtige Frage, da das Aufstellen eines Lösungsverfahrens nicht unwesentlich von dieser Klassifizierung abhängt. Liegt das betrachtete Problem in der Klasse P, lohnt es sich, eventuell sogar viel Zeit dafür zu investieren, einen guten, also polynomial begrenzten Algorithmus zu finden. Weiß man aber im voraus, daß es für das betrachtete Problem nie einen polynomialen Algorithmus geben wird, kann man von vornherein seine Kraft in das Auffinden anderer Lösungswege (insbesonderer suboptimaler) setzen.

Ein Hilfsmittel, die Zugehörigkeit eines Problems zu einer der beiden Klassen zu entscheiden, ist, das Problem auf ein anderes zu reduzieren, dessen Zugehörigkeit bekannt ist.

Def.: Es seien zwei Probleme PR_1 und PR_2 gegeben. PR_1 ist polynomial reduzierbar auf PR_2 , $PR_1 \propto PR_2$, wenn es eine Funktion f gibt, für die gilt:

1. f transformiert die Lösungen von PR_1 in Lösungen von PR_2 ;
2. es gibt einen polynomial begrenzten Algorithmus, um f zu berechnen;
3. $f(x)$ ist nur dann Lösung von PR_2 , wenn x Lösung von PR_1 ist.

Daraus folgt, wenn $PR_1 \propto PR_2$ und $PR_2 \in P$, dann ist auch $PR_1 \in P$. Diese Relation ist transitiv, d. h.

$$(PR_1 \propto PR_2 \ \& \ PR_2 \propto PR_3) \longrightarrow PR_1 \propto PR_3$$

Def.: Ein Problem PR wird NP-vollständig genannt, wenn jedes andere Problem in NP darauf polynomial reduzierbar ist, d. h. wenn

$$\forall PR_i \in NP : PR_i \propto PR$$

Somit bilden die NP-vollständigen Probleme eine Unterklasse von NP. Zu ihr gehören die härtesten Probleme in NP. Wenn für irgendein NP-vollständiges Problem ein Algorithmus mit polynomialer Zeitkomplexität gefunden werden kann, läßt

sich daraus schließen, daß es einen solchen für jedes andere Problem in NP gibt, also daß $P = NP$. Die bisherigen Erkenntnisse und Erfahrungen auf diesem Gebiet legen jedoch die Vermutung nahe, daß $P \neq NP$ ist. Diese Vermutung ist folglich äquivalent zu der Aussage, daß kein NP-vollständiges Problem in polynomialer Zeit gelöst werden kann.

Nun läßt es sich zeigen, daß viele Scheduling-Probleme NP-vollständig sind/Lag81a, Law83a/. Zu diesen gehören auch die hier interessierenden Probleme des Verteilens eines beliebigen Aufgabensystems auf ein Multiprozessorsystem. Aufgrund der Definition der NP-Vollständigkeit ist es also wenig aussichtsreich, einen polynomial begrenzten Algorithmus zu finden, der die optimale Lösung für das Problem liefert. So bleiben einem nur Algorithmen mit nichtpolynomialem Zeitverhalten. Diese beanspruchen aber mit steigender Problemgröße enorm viel Zeit. Der Rechenaufwand ist bei Problemstellungen von in der Praxis interessanter Größe schon nicht mehr vertretbar. Etwas günstigere Zeitschranken erhält man, wenn man die Suche nach optimalen Verfahren ganz aufgibt und sich mit Algorithmen begnügt, die eine Lösung finden, die der optimalen möglichst nahe kommt. Solch eine Lösung nennt man eine approximative Lösung oder Näherungslösung.

PR sei nun ein Problem derart, wie z. B. Scheduling-Probleme, I eine aktuelle Problemstellung dazu und $F_{\text{opt}}(I)$ der Wert einer optimalen Lösung für I. Ein Approximationsalgorithmus erzeugt nun eine mögliche Lösung für I, deren Wert $F_{\text{ap}}(I)$ kleiner (größer) als $F_{\text{opt}}(I)$ ist, falls PR ein Maximierungs- (Minimierungs-) Problem ist. Man kann nun verschiedene Klassen von Approximationsalgorithmen definieren/Hor81a/.

Sei A ein Algorithmus, der zu jeder Problemstellung I eines Problems PR eine mögliche Lösung erzeugt. $F_{\text{opt}}(I)$ sei eine optimale Lösung für I (immer > 0), $F_{\text{ap}}(I)$ eine von A erzeugte.

Def.: A ist genau dann ein absoluter Approximationsalgorithmus für das Problem PR, wenn für jede Problemstellung I zu PR gilt:

$$|F_{\text{opt}}(I) - F_{\text{ap}}(I)| \leq k, \text{ wobei } k \text{ eine Konstante ist.}$$

Def.: A ist genau dann ein f(n)-approximativer Algorithmus für das Problem PR, wenn für jede Problemstellung I der Größe n gilt:

$$|F_{\text{opt}}(I) - F_{\text{ap}}(I)| / F_{\text{opt}}(I) \leq f(n).$$

Def.: Ein ϵ -approximativer Algorithmus ist ein f(n)-approximativer Algorithmus, für den $f(n) \leq \epsilon$ ist, wobei ϵ eine Konstante ist.

Def.: $A(\epsilon)$ ist genau dann ein Approximationsschema, wenn $A(\epsilon)$ für jedes $\epsilon > 0$ und jede Problemstellung I eine mögliche Lösung erzeugt, so daß gilt:

$$|F_{\text{opt}}(I) - F_{\text{ap}}(I)| / F_{\text{opt}}(I) \leq \epsilon.$$

Def.: Ein Approximationsschema ist genau dann ein Approximationsschema mit polynomialem Zeitverhalten oder kurz polynomiales Approximationsschema, wenn für jedes $\epsilon > 0$ die Rechenzeit bezüglich der Problemgröße polynomial ist.

Def.: Ein Approximationsschema, dessen Rechenzeit sowohl bezüglich der Problemgröße als auch in $1/\epsilon$ polynomial ist, heißt ein volles polynomiales Approximationsschema.

Wenn sich nun schon kein polynomial begrenzter Optimierungsalgorithmus finden läßt, hätte man wenigstens gerne einen, bei dem eine konstante, maximale Abweichung vom Optimum garantiert ist, also einen absoluten Approximations-

algorithmus. Leider existieren selbst solche für NP-vollständige Probleme nicht. Ja selbst einen $f(n)$ -approximativen Algorithmus kann man für bestimmte NP-vollständige Probleme nicht finden.

Die beste Lösungsmöglichkeit, die man also für die hier interessierenden Scheduling-Probleme finden kann, ist ein Approximationsalgorithmus. Dies wird im allgemeinen ein heuristisches Verfahren sein, bei dem es aussieht, als würde es das exakte Optimum liefern. In Wirklichkeit garantiert es jedoch nur, daß die von ihm gelieferte Lösung zulässig ist und von dem Optimum nur um einen konstanten Wert oder einen multiplikativen Faktor abweicht. Solche Algorithmen hängen aufgrund ihrer heuristischen Natur zwangsläufig sehr stark von der speziellen Problemstellung ab. Deshalb ist es offensichtlich, daß man sich Ausführungsgarantien für die Rechenzeit entsprechend oben definierter Schranken wünscht, z. B. irgendwelche Schranken für die Rechenzeit, die auch im ungünstigsten Fall nicht überschritten werden ("worst-case-bounds").

Beurteilt man nun Algorithmen anhand dieser Schranken, so neigt man dazu, Verfahren als schlecht zu beurteilen, die doch üblicherweise zufriedenstellend arbeiten. In der Praxis verhalten sich Approximationsalgorithmen meist wesentlich besser, als es die "worst-case"-Schranken angeben, die doch nur das Verhalten im schlimmsten Fall beschreiben.

Daher ist eine Untersuchung, wie sich ein Algorithmus durchschnittlicherweise verhält, mindestens ebenso interessant/Co73a/. Meistens geschieht dies dadurch, daß man eine Anzahl "typischer" Beispiele von dem Algorithmus bearbeiten läßt und die Ergebnisse vergleicht. Dabei hängt die Verwendbarkeit solcher Ergebnisse sehr von der Auswahl der Beispiele ab. Wenn man über konkrete Daten eines Problems Angaben in Form irgendeiner Verteilung machen kann, ist es möglich, sich Beispiele mit zufälligen Daten, die dieser

Verteilung unterliegen, generieren zu lassen und so typische Beispiele zu erhalten. Oft ist es jedoch schwierig, eine geeignete Verteilung zu wählen.

Obwohl das Durchschnittsverhalten von Algorithmen meist wesentlich besser ist als ihr "worst-case"-Verhalten, sagt dies jedoch gar nichts darüber aus, wie sich der Algorithmus in einem speziellen vorliegenden Fall verhält, wohingegen die "worst-case-bounds" wenigstens eine Schranke für die Ausführungszeiten angeben. Deshalb sollte man, will man so viel Information wie möglich über einen Algorithmus sammeln, ihn auf beide Arten analysieren und sowohl die Schranken für das "worst-case"-Verhalten ermitteln, als auch die für das Durchschnittsverhalten.

3. Lösungsverfahren

In den letzten Jahren fand die Suche nach geeigneten Verfahren zur Lösung des geschilderten Problemkreises verstärktes Interesse. Jedoch wurden recht unterschiedliche Anstrengungen unternommen, je nachdem, ob es sich um exakte (optimale) oder heuristische (approximative, suboptimale) Lösungsverfahren, problemorientierte oder allgemeine Ansätze handelte.

Die problemorientierten Verfahren schränken den allgemeinen Ansatz für Probleme einer Klasse ein, indem sie spezifische Eigenschaften einer Unterklasse von Problemen in die Lösungsfindung mit einbeziehen. Sie sind daher in der Regel nur zur Lösung von Problemen eines speziellen Typs geeignet. Das Angebot an exakten und heuristischen Lösungsverfahren für problemorientierte Ansätze ist weitgehend ausgeglichen. Bei den allgemeinen Ansätzen überwiegen dagegen die exakten Verfahren.

Wie im zweiten Kapitel gezeigt, wird sich für das hier interessierende Problem nie ein exaktes Verfahren mit einer effizienten worst-case-Schranke finden lassen. Wohl aber zeigen die auf Optimalität abzielenden Ansätze bei bestimmten Problemen ein durchaus annehmbares Durchschnittsverhalten und sind oft auch in approximative Verfahren eingebettet. Deshalb soll zunächst einmal ein Blick auf diese Verfahren geworfen werden.

3.1. Optimale Verfahren

Um die Einordnung der im weiteren zitierten Literatur zu erleichtern, gibt die Übersicht 1 einen Überblick.

3.1.1. Lineare Programmierung

Die Verfahren der linearen Programmierung zählen wohl zu den am häufigsten verwendeten exakten Methoden. Sie zeichnen sich dadurch aus, daß man für das gerade vorliegende Problem ein mathematisches Modell in der Form eines linearen Gleichungs-/Ungleichungssystem bildet. Man definiert Variable x_i , die - allein oder kombiniert - zur Erreichung eines vorgegebenen Zieles beitragen. Für jede dieser Variablen existiert eine Funktion f_i , die darüber Auskunft gibt, welchen Beitrag x_i zur Zielerreichung liefert. Damit läßt sich dann die Zielfunktion formal so beschreiben:

$$Z(x_1, x_2, \dots, x_n) = f_1(x_1) + f_2(x_2) + \dots + f_n(x_n) \longrightarrow \max! \\ \text{(bzw. min!)}$$

Diese Zielfunktion ist nun gewissen Beschränkungen (Restriktionen, Nebenbedingungen) unterworfen, was formal wie folgt dargestellt werden kann:

$$g_i(x_1, x_2, \dots, x_n) \leq b_i \quad (1 \leq i \leq m)$$

Die linke Seite gibt dabei Auskunft darüber, in wie weit eine Alternative bestimmte Ressourcen beansprucht, die rechte Seite dagegen in welchem Maß Ressourcen des Typs i überhaupt zur Verfügung stehen. Für das betrachtete Anwendungsgebiet des Scheduling wird es üblicherweise zutreffen, daß die Variablen x_i nicht negativ sind, ja noch mehr, daß sie ganzzahlig sind. Z. B. könnte bei diesem Problemkreis der Wert einer Variablen x_i als die Nummer des Prozessors interpretiert werden, dem das Programm i zugeteilt wurde, oder als der Speicherplatzbedarf des Programms i . Für

solche Probleme, die ganzzahliger Lösungen bedürfen, gibt es nun die Möglichkeit, entweder direkt Methoden der Integer-Programmierung einzusetzen, oder die allgemeineren Verfahren der linearen Programmierung zu verwenden und die daraus resultierenden nicht ganzzahligen Ergebnisse geeignet zu runden.

3.1.1.1. Allgemeine LP-Verfahren

Die von Dantzig entwickelte Simplex-Methode/Dan66a/ ist zu einem Standard unter den LP-Verfahren geworden. Die Koeffizienten der Entscheidungsvariablen werden zu einer Matrix zusammengefaßt, die rechten Seiten der Restriktionen zu einem dazugehörenden Vektor. Aufgrund eines einfachen Tests kann die Spalte der Matrix ermittelt werden, in der das Element zu finden ist, das am meisten zu einer Verbesserung des Zielfunktionswertes beitragen würde. Ein weiterer Test ermöglicht das Lokalisieren dieses Elementes (Pivotelement). Eine neue Lösung des Gleichungs-/Ungleichungssystems entsteht nun dadurch, daß die so ermittelte Variable auf den höchstmöglichen Wert im Rahmen der Restriktion, zu der sie als Pivotvariable gehört, erhöht wird. Dann erfolgen von neuem die Tests und weitere Iterationen, bis keine Verbesserung mehr möglich ist.

Die so ermittelte optimale Lösung ist im allgemeinen nicht ganzzahlig. Ist dies gefordert, muß in einem daran anschließenden Prozeß die optimale ganzzahlige Lösung ermittelt werden. Im einfachsten Fall kann man die einzelnen Werte der Entscheidungsvariablen einfach runden. Erfahrungen zeigen aber, daß die so erhaltene ganzzahlige Lösung nicht notwendigerweise optimal sein muß. Eine Methode, das wirkliche ganzzahlige Optimum zu erhalten, ist das Schnittebenen-Verfahren von Gomory/Gom58a/, eine andere der Ansatz von Dakin/Dak65a/.

3.1.1.1.1. Schnittebenen-Verfahren

Historisch gesehen sind die Schnittebenen-Verfahren wohl der erste allgemeine Versuch zur Lösung von Integer-Programmierungsproblemen. Die Grundlagen zu dieser Modifikation der gewöhnlichen Simplex-Methode wurden durch Gomory/Gom58a/ gelegt, und mittlerweile findet man in der Literatur eine große Anzahl von Verfahren, die darauf basieren.

Die Grundidee besteht darin, zu einem mit dem Simplex-Verfahren gefundenen, nicht ganzzahligen Optimum sukzessive weitere Nebenbedingungen (Schnitte) zu erzeugen, die den nichtganzzahligen Lösungsbereich so verkleinern, daß einerseits immer alle ganzzahligen Lösungen im Lösungsbereich erhalten bleiben und andererseits nach einer endlichen Anzahl von Schnitten das Integer-Optimum durch einen Eckpunkt des verkleinerten Lösungsraums gegeben ist. Voraussetzung für die Anwendung dieser Verfahren ist die Ganzzahligkeit aller Koeffizienten und Konstanten des zu dem zu lösenden Problem gehörigen LP-Modells. Dies trifft z. B. bei den Scheduling-Problemen üblicherweise zu.

Die Anzahl an Schnitten, die zur Lösung eines aus einer praktischen Problemstellung hervorgegangenen LP-Modells erforderlich ist, ist im allgemeinen jedoch so groß, daß der Einsatz des Verfahrens nicht mehr effektiv möglich ist. Sie ist unter anderem stark von dem Auswahlkriterium abhängig, das zum Generieren neuer Schnitte herangezogen wird, sowie der momentan aktuellen, spezifischen Problemstruktur. So kann eine Problemstellung unter Umständen mehrere tausend Iterationen benötigen, während dasselbe Problem nur mit anderen Daten mit relativ wenigen Iterationen gelöst ist.

Ein weiterer wesentlicher Nachteil dieser Verfahren besteht darin, daß mit der ganzzahligen Optimallösung die erste und einzige nichttriviale Integer-Lösung überhaupt ermittelt wird. Alle vorher gefundenen Lösungen sind nicht ganzzahlig und daher als Entscheidungsalternativen für reale Probleme unbrauchbar. Damit können diese Verfahren auch

nicht bei suboptimalen Lösungsansätzen verwendet werden, bei denen man anhand eines bestimmten, zu definierenden Entscheidungskriteriums den Iterationsprozeß abbricht, z. B. weil man mit einer vorgewählten genügenden Genauigkeit an das Optimum herangekommen ist (siehe späteres Kapitel über diese Verfahren).

3.1.1.1.2. Entscheidungsbaum-Verfahren

Andere Ansätze (wie z. B. der von Dakin/Dak65a/) verwenden das mit dem Simplex-Verfahren gefundene nicht ganzzahlige Optimum als Ausgangspunkt für ein Entscheidungsbaumverfahren ("branch and bound", siehe auch Kapitel 3.1.2.), mit dem dann Schritt für Schritt das ganzzahlige Optimum ermittelt wird. Die von Geoffrion und Marsten/Geo72a/ zitierten numerischen Erfahrungen lassen jedoch darauf schließen, daß das Verfahren von Dakin zur Zeit nur zur Lösung von allenfalls mittelgroßen gemischt-ganzzahligen LP-Modellen mit nur relativ wenigen Integer-Variablen eingesetzt werden kann und somit für die rein ganzzahligen Probleme, wie sie hier vornehmlich von Interesse sind, uninteressant ist.

3.1.1.2. Integer-Programmierung

3.1.1.2.1. Allgemeines

Im allgemeinen läßt sich jedes ganzzahlige kombinatorische Problem (und auch ein Scheduling-Problem ist ja letztendlich ein solches) durch mehrere verschiedene Integer-Modelle beschreiben, die sich voneinander nur in der Größe und dem Rechenzeitbedarf unterscheiden. Jedoch sind die Algorithmen der ganzzahligen Programmierung leider meist extrem ineffizient. Der wesentliche Vorteil, den man durch das Aufstellen solcher Modelle gewinnt, ist der, daß man

einen guten Einblick in die innerere Struktur und die Größe des Problems bekommt. Kennt man aber die Struktur, dann kann man Parallelen zu anderen bekannten Problemen finden und dieselben Methoden wie zur Lösung jener anwenden oder zumindest Anstöße für die Entwicklung neuer Methoden bekommen. Spezielle Modelle der Integer-Programmierung sind jedoch auch effizient lösbar. Aber nur in wenigen Fällen wird es gelingen, allgemeine Integer-Programmierungsverfahren effizient einzusetzen. Geoffrion/Geo76a/ meint sogar, daß es niemals vollständig automatisierte LP-Lösungsverfahren für die meisten komplexen Probleme geben wird, wobei er dabei wohl das "worst-case"-Verhalten vor Augen hat. Speziell auf eine eingeschränktere Klasse von Probleme zugeschnittene Verfahren werden meist bessere Resultate liefern.

3.1.1.2.2. 0/1-Verfahren

Eine große Anzahl von Lösungsmethoden wurde für die spezielle Klasse der 0/1-Probleme entwickelt. Diese Klasse umfaßt alle die Probleme, die durch ein mathematisches Modell obengeschilderter Art beschreibbar sind, jedoch mit der Einschränkung, daß alle Variablen x_i nur die Werte 0 oder 1 annehmen können.

Auch Scheduling-Probleme lassen sich so darstellen. Als Beispiel sei das Modell für eine einfache Scheduling-Aufgabe angegeben: Ziel sei es, n Programme auf m Rechner so zu verteilen, daß die Gesamtbearbeitungszeit für alle Programme möglichst kurz ist. Die Programme seien voneinander unabhängig und alle Rechner vom gleichen Typ, d. h. alle Programme können prinzipiell auf jedem Rechner laufen und verhalten sich auf jedem gleich. Außerdem sei zur Vereinfachung des Beispiels angenommen, daß es keinerlei Engpässe bei den Ressourcen (wie Speicherplatz) gibt. Zur Modellierung werden nun Variable x_{ij} eingeführt, die den Wert 1 annehmen, wenn das Programm i dem Prozessor j zugeteilt

wird, 0 sonst. r_i sei die Rechenzeit, die das Programm i brauche. Dann sieht das Gleichungs-/Ungleichungssystem für das Problem so aus:

$$\max_j \left(\sum_{i=1}^n r_i \cdot x_{ij} \right) \longrightarrow \min!$$

$$x_{ij} \in \{0,1\}; \quad r_i \geq 0; \quad 1 \leq i \leq n; \quad 1 \leq j \leq m$$

Wohl die meisten der in der Literatur vorgeschlagenen 0/1-Verfahren sind enumerative Methoden. Sie generieren nach bestimmten Kriterien eine mögliche, erlaubte Lösung nach der anderen, bis die gerade gefundene dem Optimierungskriterium genügt und man sicher sein kann, daß es keine bessere im Sinne der Zielfunktion gibt. Die Verfahren unterscheiden sich hauptsächlich durch das Auswahlkriterium, das festlegt, welcher Pfad (bzw. welche Pfade) des Lösungsbaums durchlaufen werden, um zu dem gesuchten Optimum zu gelangen.

Eine solche Methode, einen geeigneten Weg durch den Lösungsbaum zu finden, ist die, daß man schrittweise jeweils der Lösungsvariable als nächster den Wert 1 zuweist, von der man annimmt, daß sie zur Zielerreichung am meisten beiträgt. Erkennt man, daß man in eine Sackgasse gelaufen ist, von der aus man nicht mehr zur optimalen Lösung oder gar nicht einmal mehr zu einer zulässigen gelangen kann, kehrt man zum vorherigen Knoten im Lösungsbaum zurück und verfolgt den anderen Ast weiter. Hat man auf diesem Wege eine erste zulässige Lösung gefunden, kann man diese mit als Entscheidungskriterium für das Erkennen solcher nicht weiterführender Äste verwenden.

Ein sehr bekannt gewordenes Verfahren dieser Art ist das von Hammer und Rudeanu/Ham68a/, das auch von Wunderatsch/Wun79a/ für das Scheduling eines Aufgabensystems

in einem Multiprozessorsystem eingesetzt wurde. Ein anderes viel genanntes Verfahren dieser Art ist der additive Algorithmus von Balas/Bal65a/. Er wurde Grundlage vieler weiterer Verfahren durch Modifikationen und Erweiterungen, z. B. in einem Algorithmus von Glover/Glo65a/ und der Filtermethode von Balas selbst/Bal67a/, den Verfahren von Pierce/Pie68a/ und Zionts/Zio72a/ und vielen anderen.

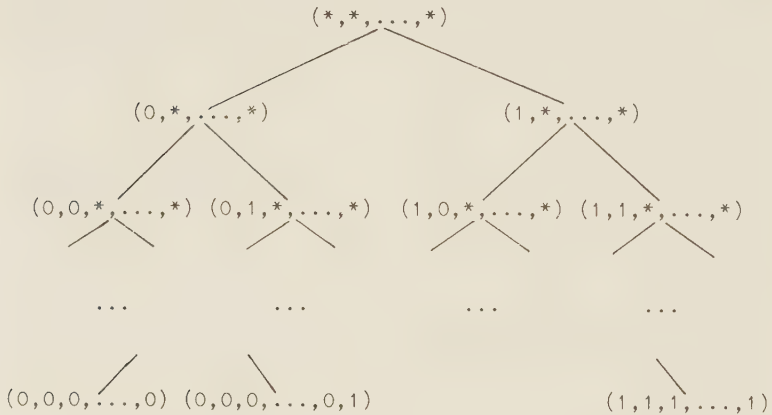


Bild 1: Beispiel eines binären Lösungsbaumes

Erläuterungen:

$(\dots) \hat{=} (x_1, x_2, \dots, x_n)$
 $*$ $\hat{=} x_i$ noch nicht festgelegt
 0 $\hat{=} x_i = 0$
 1 $\hat{=} x_i = 1$

Für bestimmte Problemstellungen liefert diese Vorgehensweise relativ schnell die optimale Lösung, so auch in den von Wunderatsch gezeigten Beispielen. Die Effizienz dieser Algorithmen hängt jedoch wesentlich von den Abbruchkriterien ab, anhand derer Teilbäume von der weiteren Betrachtung ausgeschlossen werden können. Wenn z. B. viele der Koeffizienten der Entscheidungsvariablen gleich oder nahezu gleich sind, läßt sich in den oberen Ebenen des Baumes schwer bestimmen, welcher Ast am aussichtsreichsten ist. In diesem Fall muß auf Verdacht ein Pfad weiter untersucht und häufig zum vorhergehenden Knoten zurückgekehrt werden. Mehr oder weniger werden dann doch alle Äste des Lösungsbaums abgearbeitet. Dies kommt aber wieder einem Aufzählungsverfahren gleich, kann also zu unverträglich hohen Rechenzeiten führen.

Schon Hammer selbst weist darauf hin, daß viele Probleme der realen Welt eine viel zu hohe Dimension haben, als daß sie mit Boole'schen Methoden gelöst werden könnten/Ham79a/. Es gibt eine große Klasse von 0/1-Problemen, bei denen der Lösungsraum derart umfangreich ist, daß wenig Hoffnung besteht, darin irgendeine Struktur zu entdecken/Ham74a/. Zu dieser Klasse gehören insbesondere die Probleme, bei denen alle Nebenbedingungen nur nichtnegative Koeffizienten beinhalten oder wo der Betrag der einzelnen Koeffizienten klein ist im Vergleich zu den entsprechenden rechten Seiten der Restriktionen. Gerade diese Eigenschaften können aber bei Scheduling-Problemen durchaus auftreten. Das führt dazu, daß die Rechenzeit solcher Algorithmen stark von den gerade aktuellen Eingabewerten abhängt. Als Scheduling-Algorithmus wünscht man sich aber ein Verfahren, daß, wenn es schon nicht unabhängig von den Eingabedaten ist, so doch zumindest relativ unempfindlich darauf reagiert. Auch steigt der Speicherplatzbedarf derartiger Verfahren mit zunehmender Anzahl von Variablen stark an, so daß dies für reale Probleme durchaus schon zu Schwierigkeiten führen kann.

Bei dem eben betrachteten Verfahrenstyp zur Lösung linearer 0/1-Optimierungsprobleme wird in irgendeiner Weise ein Teil des gesamten Lösungsbaumes aufgebaut. Dabei treten Rückwärtsschritte im Baum auf, wenn es sich erweist, daß der eingeschlagene Weg nicht zum Ziel führt. Dies kann unter Umständen sehr viel Zeit kosten. Ebenso tritt die Erscheinung auf, daß zwar relativ schnell zulässige Lösungen gefunden werden, die Bestätigung der Optimalität jedoch noch einen erheblichen Aufwand erfordert.

Ein anderer Lösungsansatz ist es deshalb, den Lösungsbaum ohne solche Rückwärtsschritte aufzubauen. Petrich und Wagner z. B. schlagen einen Algorithmus vor/Pet76a/, der es erlaubt, die Knoten des Baumes in aufsteigender Folge der zugehörigen Zielfunktionswerte in die Betrachtung einzubeziehen, so daß die erste ermittelte Lösung zugleich die optimale ist. Aber auch für solche Verfahren steigt der Aufwand an Rechenzeit und Speicherplatzbedarf mit der Dimension des Problems. Speziell das oben zitierte Verfahren von Petrich und Wagner ist hauptsächlich für Probleme mit relativ kleinem optimalen Zielfunktionswert geeignet. Bei der hier gestellten Problemklasse der Scheduling-Aufgaben wird man aber nicht generell sagen können, in welcher Größenordnung sich der Zielfunktionswert bewegen wird. Außerdem ist die Dimension eines realen Problems schon so groß, daß das Verfahren Speicherplatzanforderungen stellt, die durchaus schon kritisch werden können, ganz abgesehen von dem für eine aktuelle Aufgabenstellung nicht vorhersehbaren Rechenaufwand.

Wiederum ganz anders gehen z. B. Schoch und Lyska/Sch78a/ 0/1-Optimierungsprobleme an. In ihrem Algorithmus wird nicht der gesamte Lösungsbaum aufgebaut, sondern es werden grundsätzlich nur die Blätter, also die eigentlichen Lösungen, betrachtet. Auf diesen ist eine lineare Ordnung definiert, anhand derer die als nächstes zu prüfende Lösung ermittelt werden kann. Aufgrund dieser linearen Ordnung ist sichergestellt, daß jede Lösung auch

wirklich nur ein einziges Mal betrachtet wird. Zusätzliche Tests erlauben es zudem, von bestimmten, der linearen Ordnung nach eigentlich noch folgende Lösungen zu erkennen, daß sie zur Findung des Optimums nichts beitragen, und sie daraufhin aus der weiteren Untersuchung auszuschließen.

Bei diesem Algorithmus treten keinerlei Speicherplatzprobleme auf, da es keine Rückwärtsschritte gibt, und somit keine Zwischenlösungen aufbewahrt werden müssen. Wie bei allen das Optimum liefernden Verfahren ist jedoch der Rechenzeitaufwand zum Finden dieser optimalen Lösung für eine konkrete Problemstellung nicht abschätzbar. Da die lineare Ordnung anhand der Koeffizienten der Entscheidungsvariablen der Zielfunktion aufgebaut wird, wird es sich in dem Rechenzeitbedarf insbesondere bemerkbar machen, wenn viele dieser Faktoren gleich sind oder zumindest in der gleichen Größenordnung liegen.

3.1.1.3. Exkurs über NLP-Verfahren

Zur Abrundung der Darstellung der verschiedenen Verfahren soll noch einen kurzen Blick auf nichtlineare Optimierungprobleme geworfen werden, auch wenn der hier hauptsächlich interessierende Problemkreis der Scheduling-Probleme davon nicht betroffen ist.

Viele Verfahren für nicht lineare Programmierungsprobleme sind direkte Erweiterungen von LP-Methoden oder aus solchen abgeleitet/Han79a, Han69a, Sch78b/. Jedoch lassen schon geringfügige Nichtlinearitäten in der Zielfunktion oder den Nebenbedingungen die Rechenzeiten für die Verfahren immens ansteigen.

Deshalb ist es oft sinnvoll, z. B. bei Vorliegen nur geringer Nichtlinearität, diese durch Einführen einiger zusätzlicher Variablen und Restriktionen zu linearisieren. Dafür werden in der Literatur vielfältige Verfahren vorgestellt, z. B. von Bär/Bär76a/. Die nichtlinearen Optimie-

rungsprobleme werden dadurch zwar schneller lösbar, als wenn man Verfahren verwendet, die direkt auf die Nichtlinearitäten abgestimmt sind, es bleiben aber trotzdem die Schwierigkeiten bestehen, die die LP-Verfahren mit sich bringen. Man kann dann zwar sagen, daß die Lösung dieser Probleme nicht mehr ganz so viel Rechenzeit braucht, wie bei Verwendung von Methoden der nichtlinearen Programmierung, trotzdem ist jedoch der Rechenzeit- und Speicherplatzbedarf nicht im voraus exakt abschätzbar, und kann im Extremfall doch noch so groß werden, daß es für praktische Probleme untragbar ist.

3.1.2. Enumeration

Die enumerativen Methoden listen alle möglichen Lösungen des gegebenen Problems auf und eliminieren dabei diejenigen, die sich im Verlauf dieses Enumerierens als nicht optimal erweisen.

Die einfachste dieser Vorgehensweisen ist die explizite Enumeration, auch vollständige Enumeration genannt. Unabhängig von jedem Modell der linearen Programmierung können kombinatorische Probleme mittels dieser Verfahren angegangen werden. Das Prinzip dabei ist, alle zulässigen Lösungen zu bestimmen und die optimale durch den Vergleich der Zielfunktionswerte zu ermitteln. Der Vorteil der expliziten Enumeration gegenüber Integer-Programmierungsmodellen, Baumsuch-Verfahren usw. ist die Einfachheit dieser Methode. Andererseits ist die effiziente Anwendung der vollständigen Enumeration durch die Anzahl von Schritten begrenzt, die man benötigt, um alle möglichen Lösungen zu generieren. Deshalb muß diese Anzahl von Schritten im voraus bekannt sein, um entscheiden zu können, ob explizite Enumeration eine geeignete Methode zur Lösung des anstehenden Problems ist oder nicht.

Betrachtet man ein Scheduling-Problem, in dem n Programme auf m Prozessoren verteilt werden sollen, so existieren da prinzipiell m^n mögliche Lösungen. Diese Zahl kann im Einzelfall durch Restriktionen etwas beschränkt werden. Jedoch sieht man, daß der Rechenaufwand für das Auffinden des Optimums mittels expliziter Enumeration mit wachsender Problemgröße ins Unermeßliche steigt. Schon für ein Problem mit drei Prozessoren und zwanzig Prozessen sind im Extremfall annähernd $3,5 \cdot 10^9$ Lösungen zu überprüfen, was nicht mehr in angemessener Zeit durchführbar ist. Noch dramatischer wird die Situation, wenn es nicht nur auf eine einfache Zuordnung der Programme zu den Prozessoren ankommt, sondern auch die Reihenfolge auf den einzelnen Prozessoren eine Rolle spielt.

Wegen der unbefriedigenden Verwendbarkeit dieser Enumerationsmethode sollen nun die differenzierteren Verfahren der impliziten Enumeration, auch Baumsuch-Verfahren genannt, betrachtet werden. Diese Vorgehensweise unterscheidet sich von der expliziten Enumeration dadurch, daß in der Logik zwar implizit auch die Generierung aller Lösungen enthalten ist, daß jedoch keineswegs jede Möglichkeit explizit betrachtet wird, sondern daß man einen Zweig des Lösungsbaumes nicht mehr weiter verfolgt, sobald man feststellt, daß er die gesuchte optimale Lösung nicht enthalten kann.

Die Schwierigkeit in der Entwicklung von impliziten Enumerationsmethoden für bestimmte Probleme liegt darin, daß trotz dieser intelligenteren Vorgehensweise immer noch eine ungeheuer große Anzahl von Ästen abgeschnitten werden muß. Je früher deshalb erkannt werden kann, daß ein Ast nicht zum Ziele führt, desto effizienter ist der Algorithmus für diese Problemstellung.

Zur Veranschaulichung sei noch einmal das Zuordnungsproblem von n Prozessen auf m Prozessoren aufgegriffen. Die Gesamtzahl der Knoten im Lösungsbaum ergibt sich als Summe

der Knoten auf den einzelnen Ebenen. Nimmt man an, daß die Zuordnung eines bestimmten Programms zu einem Prozessor einer Ebene entspricht, und jeder Knoten dieser Ebene so viele Nachfolger hat, wie Prozessoren für die Zuteilung der nächsten Aufgabe zur Verfügung stehen, dann hat der gesamte Enumerationsbaum

$$\sum_{i=0}^n m^i$$

Knoten. Das sind bei drei Prozessoren und zwanzig Prozessen 5.230.176.601 Knoten. Selbst wenn man durch einen guten impliziten Enumerationsalgorithmus einen sehr hohen Prozentsatz der Äste abschneiden kann, bleiben doch noch ungeheuer viele Zweige, die überprüft, und Lösungen, die explizit ermittelt werden müssen, zumal interessante Probleme der Wirklichkeit eine durchaus höhere Dimension haben.

Umso erstaunlicher ist die Tatsache, daß es durchaus größere Probleme gibt, die durch Baumsuch-Verfahren effizient lösbar sind. Hingegen existieren auch wesentlich kleinere Problemstellungen, für die es keine implizite Enumerationsmethode gibt, die die optimale Lösung innerhalb angemessener Zeit bestimmen kann. Da die Effizienz dieser Verfahren von Problemkreis zu Problemkreis variiert, ja sogar von einer spezifischen Problemstellung zur anderen, ist es wichtig, den Aufbau impliziter Enumerationsverfahren genau zu kennen, um für die aktuell vorliegende Aufgabenstellung die geeigneteste Methode herauszufinden bzw. entwickeln zu können.

Eine gute Übersicht über die Morphologie impliziter Enumerationsverfahren findet sich bei Müller-Merbach/Mül75a/, aufgeteilt nach genereller Organisation des Enumerationsbaumes, generellem Verzweigungsprinzip, Anzahl der Verzweigungen pro Knoten, Anzahl der zum Verzweigen herangezogenen Entscheidungsvariablen, Zielfunktion, Grenzwertberechnung, Abbruchkriterien usw.

Im Rahmen dieser Arbeit sollen nicht alle diese Details betrachtet, sondern nur noch etwas die drei grundlegenden Prinzipien der Baumsuch-Verfahren beleuchtet werden.

3.1.2.1. "Branch and Exclude"

Die Methode der begrenzten Enumeration/Düc77a, Mül75a/, für die man in der Literatur auch die Begriffe "Backtracking"/Hor81a/, "Branch and Exclude"/Bal70a/ und andere findet, verwendet das gleiche sequentielle Konstruktionsprinzip des Lösungsbaumes wie die vollständige Enumeration. Wie dort ist in der Logik zwar implizit die Generierung aller Lösungen enthalten, jedoch bricht man die weitere Bearbeitung eines Astes ab, wenn die Unzulässigkeit der sich dabei ergebenden Lösungen im voraus erkennbar ist, und schließt somit etliche Lösungen von der expliziten Betrachtung aus.

Für ein Scheduling-Problem könnte diese Vorgehensweise z. B. bedeuten, daß man eine bestimmte Betrachtungsreihenfolge für die Prozessoren festlegt. Jedesmal, wenn das nächste Programm einem Prozessor zugeordnet werden soll, versucht man es zuerst bei dem ersten Prozessor in dieser Reihenfolge. Ist die Zuteilung dort z. B. wegen bestimmter Restriktionen nicht möglich, betrachtet man den nächsten in der vorgegebenen Reihenfolge usw. bis zum letzten. Kann Programm k keinem Prozessor zugeteilt werden, wird Programm k-1 von seiner jetzigen Position zum in der Reihenfolge nächsten Prozessor verlegt. Daraufhin versucht man erneut Programm k zuzuordnen. Ist dies immer noch nicht möglich, muß Programm k-1 noch einen Prozessor weiter wandern. Ist es beim letzten in der Reihenfolge angelangt und ist es immer noch nicht gelungen, Programm k unterzubringen, muß nun zu Programm k-2 zurückgekehrt werden ("backtracking"). Hat man mit dieser Vorgehensweise eine erste zulässige Lösung gefunden, so kann man den zugehörigen Zielfunktionswert als zusätzliche Restriktion verwenden. Spätere Teil-

lösungen, bei denen dieser Wert schlechter im Sinne der Zielfunktion ist, brauchen nicht weiter verfolgt werden.

Dieser Algorithmus ist jedoch trotz der gewissen Begrenzung immer noch NP-vollständig und kann daher sehr viel Zeit benötigen, um überhaupt brauchbare Lösungen oder gar die optimale zu erzeugen.

3.1.2.2. "Branch and Bound"

Es existiert keine allgemein anerkannte Terminologie für die Klasse von Methoden, die auch als "Branch and Bound"-Algorithmen bekannt geworden sind. Ein Überblick über die gebräuchlichen Begriffe und Unterschiede bei diesen Verfahren kann man bei Balas und Guignard/Bal79a/, French/Fre82a/ oder Lawler und Wood/Law66a/ finden. Im allgemeinen ist die "Branch and Bound"-Technik sowohl auf lineare Probleme anwendbar, als auch auf nicht lineare.

3.1.2.2.1. Grundlegende Prinzipien

Diese Methoden arbeiten ähnlich der begrenzten Enumeration, d. h. implizit ist in der Logik die Generierung aller Lösungen enthalten, jedoch wird ein Zweig des Lösungsbaumes nicht weiter verfolgt, sobald man feststellt, daß er die gesuchte optimale Lösung nicht enthalten kann. Anders als bei der begrenzten Enumeration hingegen wird bei der Auswahl des nächsten zu betrachtenden Knotens nicht einfach der nächste in irgendeiner einmal festgelegten Reihenfolge genommen, sondern mit Hilfe intelligenterer Algorithmen versucht, den für die weitere Bearbeitung geeignetesten herauszufinden.

Alle diese Methoden lösen die Probleme, indem sie die Menge der zulässigen Lösungen in Untermengen aufspalten, Schranken für den Zielfunktionswert für jede dieser Unter-

mengen berechnen und anhand dieser Grenzwerte bestimmte Mengen von der weiteren Betrachtung ausschließen. Die einzelnen Mengen werden weiter und weiter aufgespaltet, bis schließlich jede Menge nur noch eine einzige Lösung enthält.

Jeder "Branch and Bound"-Algorithmus beinhaltet drei wesentliche Komponenten:

1. die Suchstrategie:

Mit Hilfe der Suchstrategie wird die nächste zu betrachtende Untermenge von Lösungen ausgewählt, d. h. der nächste Knoten im Enumerationsbaum. Man kann dabei prinzipiell zwei Strategien unterscheiden:

- a) Die eine, bekannt als "breadth first", "frontier search" oder "branch from lowest bound", wählt unter allen Knoten immer denjenigen aus, der den besten unteren Grenzwert besitzt.
- b) Die andere, mit Namen "depth first" oder "newest active node", sucht in erster Linie den besten Nachfolger des aktuellen Knoten, falls ein solcher vorhanden ist, und kehrt andernfalls zum Vorgängerknoten zurück, bei dem dann erneut diese Regel angewendet wird.

Beide Strategien haben ihre Vor- und Nachteile. Während die zweite weniger Speicherplatz benötigt, betrachtet die erste im allgemeinen weniger Knoten des Baumes, braucht dadurch weniger Rechenzeit und Speicherplatz wie die erste und findet aufgrund des intelligenteren Suchverfahrens schneller das Optimum. Die beste Suchstrategie wird aber wahrscheinlich irgendwo zwischen diesen Extrempunkten liegen, etwa so, daß immer der beste Nachfolgerknoten ausgewählt wird, aber daß nicht automatisch der zuletzt betrachtete Vorgänger wieder an die Reihe kommt, wenn es keinen Nachfolger mehr gibt.

2. die Verzweigungsregel:

Die Verzweigungsregel wird dazu benutzt, die Lösungsmenge in geeignete Untermengen zu zerlegen, d. h. den Enumerationsbaum aufzubauen und die Nachfolger und Vorgänger der einzelnen Knoten festzulegen. Zieht man zum Verzweigen nur eine einzige Entscheidungsvariable heran, hat man die Auswahl unter einer Vielzahl heuristischer Regeln. Je mehr Variablen jedoch in die Verzweigungsregel mit einbezogen werden können, desto effizienter erweist sich diese in der Praxis.

3. der Algorithmus zur Berechnung der Schranken:

Die wichtigste Komponente aller Enumerationsmethoden hinsichtlich ihres Einflusses auf die Effizienz ist wohl der Algorithmus zur Berechnung der Schranken. Durch kleine Schranken werden mehr Äste frühzeitig abgeschnitten, so daß sich die gesuchte Optimallösung schneller erreichen läßt. Zur Ermittlung dieser Schranken finden Schnittebenen-Verfahren (siehe Kapitel 3.1.1.) Anwendung, ebenso wie Lagrange-Multiplikatoren, auf linearen Programmen aufbauende logische Tests und die verschiedensten, problemorientierten Heuristiken. Garfinkel gibt in seinem Buch/Garf72a/ einige Möglichkeiten und zusätzliche Tests an, durch die man häufig schneller und mehr Äste des Lösungsbaumes abschneiden kann.

Eines der Hauptmerkmale vieler "Branch and Bound"-Prozeduren ist, daß ihr Verhalten bezüglich der Rechenzeit nicht vorhersagbar ist. Der insgesamt benötigte Rechenzeitaufwand hängt von der Anzahl an generierten Untermengen von Lösungen ab und somit von der Gesamtzahl an Knoten im Enumerationsbaum. Im schlechtesten Fall können sich die Verfahren wie die der expliziten Enumeration verhalten. Der zu jedem Zeitpunkt benötigte temporäre Speicherplatzbedarf richtet sich nach der maximalen Größe einer der aktuell zu betrachtenden Untermengen, also nach der maximalen Anzahl an Blättern im Baum, die bei irgendeinem Zwischenstadium zur Betrachtung anstehen. In den verschiedenen "Branch and

Bound"-Verfahren werden die verschiedensten Methoden vorgeschlagen, diese Anzahl an Informationen, die noch im Speicher gehalten werden muß, zu begrenzen. Auch läßt sich der Rechenzeitbedarf auf Kosten von zusätzlicher Speicherplatzanforderungen senken und umgekehrt.

Allgemein gesehen reicht die Gruppe der "Branch and Bound"-Algorithmen von gänzlich heuristisch orientierten Verfahren, die nur der elementarsten mathematischen Grundlagen bedürfen, bis hin zu theoretisch sehr ausgefeilten Methoden, die auf Gruppentheorie und dergleichen beruhen. Man kann die strukturellen Eigenschaften eines Problems auf viele verschiedene Arten in Algorithmen realisieren. Daher braucht es viel Erfahrung und eine große Anzahl von Tests, um entscheiden zu können, welcher Lösungsweg gerade der effizienteste ist bzw. ob es für das anstehende Problem überhaupt einen effizienten Algorithmus gibt. Denn ein Verfahren, das für eine bestimmte Klasse von Problemen effizient ist, ist dies oft keineswegs für andere.

3.1.2.2.2. Einige Verfahren

Schon im vorangegangenen Abschnitt über Integer Programmierungsverfahren wurden einige Möglichkeiten aufgezeigt, die man ebenso gut auch hier hätte einreihen können. Es sind dies die auf 0/1-Variablen beruhenden Methoden von Hammer und Rudeanu/Ham68a/ und Wunderatsch/Wun79a/, aber auch der additive Algorithmus von Balas/Bal65a/ und die darauf aufbauenden Verfahren von Glover/Glo65a/, Balas/Bal67a/, Pierce/Pie68a/, Zions/Zio72a/ und andere. Diesen Verfahren liegt ein binärer Entscheidungsbaum zugrunde, d. h. jeder Knoten hat genau zwei Nachfolger. Üblicherweise wird dabei jede Ebene im Baum einer Variablen x_i zugeordnet. Die von jedem Knoten einer Ebene ausgehenden Pfade entsprechen den Bedingungen $x_i = 0$ und $x_i = 1$ (siehe auch Bild 1). Hansen/Han79a/ gibt noch einige weitere Möglichkeiten an, in einem binären Lösungsbaum zu

verzweigen, z. B. indem man sich zwei Variablen x_j und x_k herausgreift, und zwei neue Äste generiert entsprechend den Bedingungen $x_j = x_k$ und $x_j = 1 - x_k$ oder $x_j = 1$ und $x_k = 0$.

Generell gesehen erreichen "Branch and Bound"-Verfahren bei der Lösung von 0/1-Problemen häufig eine recht gute Leistungsfähigkeit, mehr als bei allgemeineren Entscheidungsbäumen. Viele Autoren verstehen unter "Branch and Bound"- und impliziten Enumerationsmethoden jedoch nur solche, denen Variablen zugrundeliegen, deren Wertebereich nicht auf Null und Eins beschränkt ist. Daher wurden auch die obigen Verfahren im Abschnitt über Integer-Programmierung näher beschrieben. Jetzt sollen noch einige andere Verfahren angesehen werden.

Ein simpler "Branch and Bound"-Algorithmus wurde von Charlton und Death/Cha70a/ für allgemeine Scheduling-Probleme vorgestellt. Bei ihnen entsprechen die Knoten Teillösungen, bei denen einige Programme schon bestimmten Prozessoren zugeordnet sind, andere müssen noch betrachtet werden. Die i -te Ebene im Baum entspricht der Zuweisung des i -ten Programms. Dementsprechend hat jeder Knoten der vorhergehenden Ebene so viele Nachfolgeknoten, wie es Prozessoren gibt, die das i -te Programm ausführen können. In dem in Bild 2 dargestellten Fall können beispielsweise Programm A, B und D allen Prozessoren zugeordnet werden, wohingegen Programm C nur auf Prozessor 2 laufen kann. Die letzte Ebene entspricht vollständigen, zulässigen Lösungen.

Der Algorithmus generiert nun systematisch diese zulässigen Lösungen, indem er sich von Ebene zu Ebene den Baum hinunter hangelt. Ist eine erste zulässige Lösung gefunden, wird der erreichte Zielfunktionswert als Schranke in die weitere Auswahl von Pfaden mit einbezogen. Dadurch werden Teilbäume ignoriert, die nur Lösungen mit einem schlechteren Zielfunktionswert liefern können. Kann man bereits zu Anfang eine Schranke für den Zielfunktionswert angeben, kann diese schon zur Ermittlung der ersten

zulässigen Lösung mit einfließen. Es können somit weitere Teilbäume von der Betrachtung ausgeschlossen werden.

Charlton und Death selbst geben jedoch als Kritik für diese sehr "straight-forward"-Methode an, daß sie zu jedem Zeitpunkt eine sehr große Menge an Information im Speicher halten muß. Wie schon im letzten Abschnitt erwähnt, kann der Speicherplatzbedarf dadurch verringert werden, daß ein anderer Suchalgorithmus zum Auffinden des nächsten zu betrachtenden Knotens im Baum verwendet wird. Die Zeit, die zum Ermitteln der optimalen Lösung benötigt wird, ist endlich, kann aber bei Problemen mit einer größeren Anzahl von Programmen und Prozessoren doch recht lang sein.

Dakin/Dak65a/ gibt einen Algorithmus an, der von einer kontinuierlichen Lösung des Problems ausgeht. Haben bei dieser alle Variablen ganzzahlige Werte, hat man damit die endgültige Lösung schon gefunden. Andernfalls wird die Integer-Lösung davon ausgehend mittels eines Baumsuch-Verfahrens ermittelt. Dabei wird der Lösungsbaum so aufgebaut, daß jeder Knoten - wie bei den auf 0/1-Variablen beruhenden Methoden - nur zwei Nachfolger hat. Auch hier entspricht jede Ebene im Baum der Festlegung einer Variablen. Hat diese Variable x_i einen kontinuierlichen Wert zwischen den ganzzahligen Werten k und $k+1$, so enthält der eine Zweig der nachfolgenden Ebene alle Lösungen, in denen die Variable $\leq k$ ist, der andere alle, in denen sie $\geq k+1$ ist. Ist die Variable x_i schon in der Ausgangslösung ganzzahlig mit Wert k , erhält man die Äste $x_i < k$ und $x_i > k$.

Prinzipiell gibt es für jedes Problem nicht nur einen Baum, da man in jeder Ebene die Freiheit hat, eine beliebige der noch offenstehenden Variablen auszuwählen. Die Wahl der Variablen hat einen nicht geringen Einfluß auf die vom Algorithmus benötigte Rechenzeit. Die Erfahrung von Dakin mit diesem Verfahren zeigt, daß die Auswahl der Variablen, die den Lösungswert am meisten in Richtung des gesuchten Optimums beeinflusst, am schnellsten zum Ziele führt.

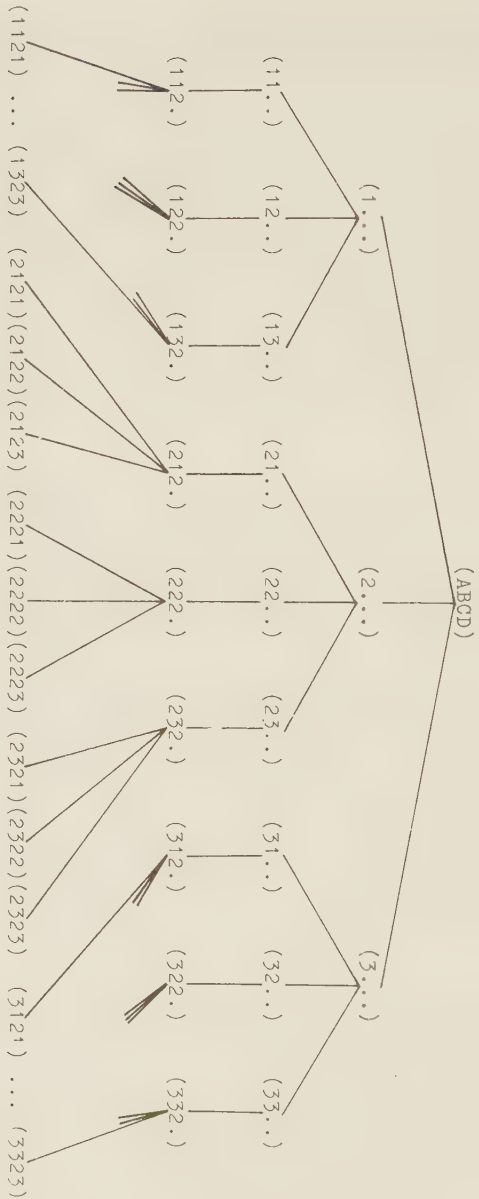


Bild 2: Beispiel eines allgemeinen Lösungsbaumes für die Zuordnung von 4 Programmen (A,B,C,D) zu 3 Prozessoren (1,2,3)

Erläuterungen:

- $\hat{=}$ Prozessorzuordnung noch nicht festgelegt
- $n \hat{=}$ Programm i ist Prozessor n zugeordnet
- (i ergibt sich aus der Stellung von n innerhalb des Programmvektors)

Ebenfalls auf einem nicht ganzzahligen Ansatz beruht das Verfahren von Greenberg/Gre68a/ zur Lösung von Scheduling-Problemen beliebiger Art. Er geht von einem allgemeinen LP-Modell aus. Aus der Lösung sucht er sich eine Alternative aus, d. h. eine Kombination Programm - Prozessor, die noch nicht in der Lösung enthalten ist, und generiert zwei neue LP-Probleme durch hinzufügen einer Restriktion, durch die eine ganzzahlige Lösung für eine weitere Variable erzwungen wird. Dieser Prozeß wird so lange fortgesetzt, bis eine vollkommen ganzzahlige Lösung vorliegt, die dann zugleich optimal ist.

Da sich dieses Verfahren ganz auf die lineare Programmierung stützt, trifft alles dort gesagte auch auf diese Methode zu, d. h. der Speicherplatz- und Rechenzeitaufwand kann sehr stark ansteigen. Als Abschätzung für sein Verfahren gibt Greenberg an, daß die approximative lineare Relation zwischen Rechenzeitbedarf und Problemgröße als $n!^m$ (n = Anzahl der Programme, m = Anzahl der Prozessoren) definiert werden kann. Das ist eine Größe, die die Anwendung dieses Algorithmusses allenfalls auf sehr kleine Probleme sinnvoll erscheinen läßt.

Zusammenfassend seien noch einmal kurz die wesentlichen Eigenschaften und Merkmale der impliziten Enumerationsverfahren genannt:

- a) Sie haben meist eine einfachere Arithmetik, aber eine komplizierte Logik als z. B. die Schnittebenen-Verfahren und sind dementsprechend meist mit einem höheren Rechenaufwand als jene verbunden.
- b) Recht gute Resultate hat man teilweise mit "Branch and Bound"-Verfahren gemacht, die auf binären Entscheidungsbäumen aufbauen. Jedoch betrachten die Verfahren oft nur die Zielfunktion oder eine Restriktion zur selben Zeit, wodurch die Effizienz doch sehr beeinträchtigt wird.

- c) Die Entscheidungsbaum-Methoden liefern häufig recht rasch eine brauchbare zulässige Lösung (oder sogar die optimale), brauchen aber danach noch unverhältnismäßig lange, um diese zulässige Lösung zu verbessern oder gar die Optimalität nachzuweisen.
- d) Nicht zu übersehen ist der erhebliche Speicheraufwand, der im allgemeinen den Enumerationsverfahren anhaftet und der in den meisten Fällen exponentiell mit dem Umfang des Problems wächst.
- e) Die Güte des Verfahrens hängt nicht nur vom Umfang des Problems ab, sondern ganz wesentlich auch von den Zahlenwerten der Konstanten, die im Problem auftreten. Zwei Probleme, die die gleiche Struktur aufweisen, gleichviel Variablen und gleichviel Zahlenkonstante besitzen und sich nur durch die Werte der Zahlenkonstanten unterscheiden, werden mittels desselben "Branch and Bound"-Verfahrens in unterschiedlicher Zeit gelöst, weil unterschiedlich viele Iterationen bis zur vollständigen Lösung ausgeführt werden müssen.
- f) Viele Tests zeigen, daß diese Verfahren meist nur bei sehr speziellen Problemstellungen sinnvoll einsetzbar sind, z. B. dort, wo die Restriktionen den gesamten potentiellen Lösungsraum schnell einschränken.

3.1.2.3. Dynamische Programmierung

Die Methode der dynamische Programmierung ist das älteste Entscheidungsbaum-Verfahren. Sie unterscheidet sich von den bisher besprochenen Ansätzen in der Organisation des Enumerationsprozesses.

Die dynamische Optimierung verwendet eine streng parallele Organisation des Enumerationsprozesses. Die einzelnen Äste des Lösungsbaums werden parallel zueinander aufgebaut. In jeder Stufe des Prozesses werden die den parallelen

Knoten entsprechenden Teillösungen verglichen. Die Speicheranforderung des Algorithmusses ergibt sich daher aus der Anzahl der parallelen Knoten. Deshalb kann die dynamische Programmierung in allen den Fällen eine geeignete Methode sein, wo der Enumerationsbaum tief und schmal ist, d. h. viele Nachfolgeknoten in jedem Pfad vorhanden sind, aber nur wenige parallele Knoten in jeder Ebene. Im Gegensatz dazu ist "Branch and Bound" überall dort sinnvoll, wo ein breiter Lösungsbaum mit vielen parallelen Knoten vorliegt, bei dem aber die einzelnen Pfade nicht allzu tief reichen, da dort alle Zwischenlösungen auf einem Pfad zwischengespeichert werden müssen und es daher bei vielen Nachfolgern zu Speicherplatzproblemen kommen kann.

Die dynamische Programmierung ist eine Methode zum Entwurf von Algorithmen, die man dann anwenden kann, wenn die Lösung eines Problems als Ergebnis einer Folge von Entscheidungen angesehen werden kann. Man geht davon aus, daß das vorliegende Problem in Teilprobleme zerlegbar ist und daß man eine optimale Folge von Entscheidungen dadurch erreichen kann, daß man eine Entscheidung nach der anderen trifft und niemals einen Fehler macht, d. h. niemals eine zu einem früheren Zeitpunkt getroffene Entscheidung rückgängig machen muß ("backtracking").

Beim Scheduling von n Programmen könnte das z. B. die Zuordnung von $n-1$, $n-2$, $n-3$ Programmen usw. sein. Für jede dieser Teilzuordnungen müßte bestimmbar sein, ob sie zum Optimum führt oder nicht. Die Vorgehensweise ist also einfach. Man ordnet zuerst ein Programm optimal zu, sucht von dieser Lösung ausgehend die beste Zuteilung für das zweite Programm, dann für das dritte usw. bis alle Programme zugeordnet sind. Kann zu einem Zeitpunkt das nächste anstehende Programm keinem Prozessor zugeteilt werden, bricht der Algorithmus ab, da dann feststeht, daß es auch dann keine Lösung geben wird, wenn man die bisher schon betrachteten Programme anders zuordnen würde.

Beim dynamischen Programmieren erreicht man eine optimale Entscheidungsfolge, indem man sich explizit auf das Optimalitätsprinzip beruft.

Optimalitätsprinzip:

Eine optimale Entscheidung hat die Eigenschaft, daß unabhängig vom Anfangszustand und von der Anfangsentscheidung die übrigen Entscheidungen eine optimale Entscheidungsfolge bilden müssen unter Berücksichtigung des aus der ersten Entscheidung resultierenden Zustands.

Das Optimalitätsprinzip ist nur im Hinblick auf den Anfangszustand und die erste Entscheidung formuliert, kann aber ebenso gut auf Zwischenzustände und -entscheidungen angewendet werden. Diese wiederholte Anwendung führt zu einer Rekursionsformel. Algorithmen, die auf dem Prinzip des dynamischen Programmierens beruhen, lösen diese Rekursion und erhalten so die Lösung des gegebenen Problems. Bei der Formulierung dieser Rekursionsformel kann man auf zwei verschiedene Arten vorgehen. Seien $x_1, x_2, \dots, x_n$ die Variablen, bezüglich derer eine Folge von Entscheidungen bestimmt werden muß. Bei der Vorwärtsmethode wird die Entscheidung für x_i in Form optimaler Entscheidungsfolgen für $x_{i+1}, \dots, x_n$ formuliert. Bei der Rückwärtsmethode hingegen wird die Entscheidung für x_i in Form optimaler Entscheidungsfolgen für $x_1, \dots, x_{i-1}$ formuliert. Bei vielen Problemstellungen ist es einfacher, die Rekursionsbeziehung mit Hilfe der Rückwärtsmethode zu erhalten.

Durch die Vorgehensweise der dynamischen Programmierung können durchaus viele Entscheidungsfolgen erzeugt werden. Jedoch können Folgen, die suboptimale Teilfolgen enthalten, nicht optimal sein, falls das Optimalitätsprinzip gilt. Daher werden sie - so weit wie möglich - nicht generiert.

Die Gesamtzahl verschiedener Entscheidungsfolgen hängt exponentiell von der Anzahl möglicher Entscheidungen ab, d. h. bei n Entscheidungen und je d Alternativen dafür hat man unter d^n Entscheidungsfolgen die optimale zu bestimmen. Algorithmen, die dynamische Programmierung verwenden, haben dagegen oft nur polynomiale Komplexität.

Die dynamische Programmierung ist jedoch nur dort einsetzbar, wo das Optimum eines Teilproblems zum Gesamtoptimum führt. Bei vielen Problemen ist es hingegen nicht möglich, schrittweise Entscheidungen, die nur auf lokalen Informationen beruhen, so zu treffen, daß die gesamte Entscheidungsfolge optimal wird. Dies ist auch bei den meisten Scheduling-Problemen der Fall. Hat man z. B. das Ziel vor Augen, eine möglichst gleichmäßige Auslastung der Prozessoren zu erreichen, so nützt einem eine bezüglich der Rechenzeit totale Gleichverteilung der ersten Hälfte der Programme gar nichts, wenn dadurch die Zuteilung der zweiten Hälfte der Programme z. B. wegen Speicherbeschränkungen völlig unmöglich gemacht wird.

Eine weitere Schwierigkeit ist, wie bei den "Branch and Bound"-Verfahren, daß Zwischenergebnisse für weitere Berechnungen gespeichert werden müssen. So begrenzt die zur Verfügung stehende Speicherkapazität auch hier die Komplexität der Probleme, die noch lösbar sind. Es hat sich gezeigt, daß auch mittels eines Ansatzes der dynamischen Programmierung nur Problemstellungen mit relativ kleiner Komplexität handhabbar sind/Fre82a/.

3.1.3. Graphentheoretische Verfahren

Ein weiterer in der Literatur weitverbreiteter Ansatz zur Lösung von Zuordnungsproblemen sind graphentheoretische Verfahren. Bei diesen Methoden werden die Prozessoren und Programme als Knoten eines Graphen aufgefaßt; die Kommunikationsstruktur bildet die Kanten zwischen den Knoten. Man

orientiert sich in den meisten Fällen an zwei Kostenarten, die als Entscheidungskriterium zur Erreichung einer guten Prozeßzuordnung in die Zielfunktion einfließen:

1. Jeder Prozeß verursacht Rechenkosten, die in einem heterogenen Prozessorsystem von den Leistungsmerkmalen der einzelnen Prozessoren (wie z. B. Geschwindigkeit, Befehlsvorrat oder ähnlichem) abhängen.
2. Ein zusätzlicher Aufwand entsteht, wenn zwei miteinander kommunizierende Prozesse unterschiedlichen Prozessoren zugeteilt werden. In diese Kommunikationskosten geht vor allem die ausgetauschte Informationsmenge ein. Abgeleitet von der Systemstruktur können aber auch Eigenschaften der Kommunikationswege, wie z. B. Entfernungen, miteinfließen.

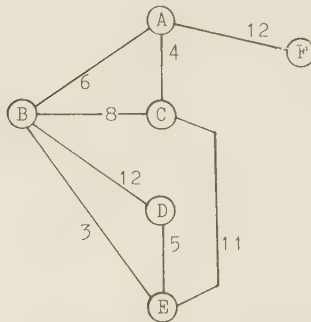


Bild 3: Programmgraph

Die Graphentheorie bietet zur Optimierung solcher Zuordnungsgraphen das Netzflußkonzept ("network-flow-concept") an. Man entwickelt einen Programmgraphen (Bild 3), in dem die Knoten den Programmen entsprechen, die Kanten den Kommunikationsverbindungen, beschriftet mit den Kommunikationskosten. Prozesse, die keine Informationen miteinander austauschen, haben die Kommunikationskosten Null; die Kante entfällt im Graphen. Ist eine Kante mit ∞ beschriftet, so bedeutet das, daß die beiden Prozesse auf jeden Fall dem gleichen Prozessor zugeordnet werden müssen.

Um nun auch die Rechenkosten in den Graphen einfließen lassen zu können, erweitert man den Programmgraphen um Knoten, die den Prozessoren entsprechen (Bild 4). Von ihnen führen Kanten zu jedem Programmknoten, beschriftet mit den jeweiligen Rechenkosten, die der Lauf des Programms auf diesem Prozessor verursachen würde. Eine Beschriftung von ∞ heißt in diesem Fall, daß dieses Programm auf keinen Fall auf diesem Prozessor laufen darf.

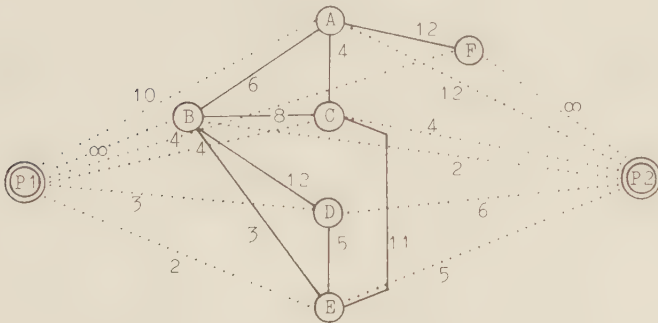


Bild 4: modifizierter Programmgraph

Mit einem Netzfluß-Algorithmus kann nun die Leistungsfähigkeit eines beliebig vermaschten Netzes - wie dies der so erhaltene modifizierte Programmgraph sein kann - bestimmt werden. Sie ergibt sich aus dem Datenfluß, der zwischen den Quell- und Zielknoten des Netzes möglich ist. Ford und Fulkerson haben das bekannte "Max-Flow/Min-Cut-Theorem" formuliert, das Grundlage jedes Netzfluß-Algorithmusses ist:

Max-Flow/Min-Cut-Theorem:

Der maximale Fluß in einem Netzwerk ist gleich der Kapazität des minimalen Schnittes (minimal cut) in einem Netzwerk.

(Beweis siehe /For62a/)

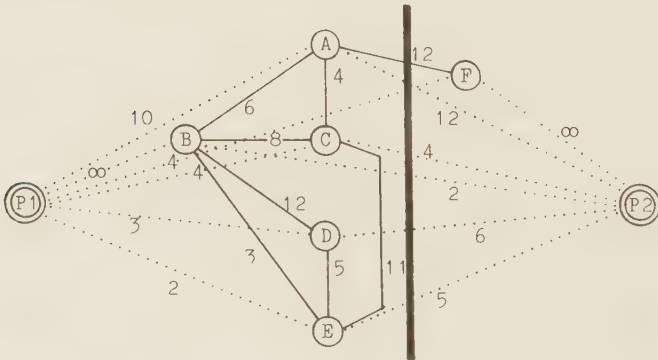


Bild 5: modifizierter Programmgraph mit minimalem Schnitt

Ein Schnitt in einem Netzwerk ist eine Menge von Kanten, die den Graphen in zwei Teilgraphen zerlegen, wenn man sie entfernt. Dabei dürfen Quell- und Zielknoten nicht in

ein und demselben Teilgraphen liegen. Das von Stone/Sto77a/ formulierte Theorem über die Äquivalenz eines Schitts (cut) und einer Prozessorzuordnung besagt, daß die Prozeßzuordnungskosten mit der Kantengewichtssumme des entsprechenden Schnitts identisch sind. Da eine optimale Lösung für das Zuordnungsproblem erwünscht ist, d. h. eine mit minimalen Kosten, muß die Kantenschnittmenge mit der geringsten Gewichtssumme (minimal cut) gesucht werden. Die Quell- und Zielknoten entsprechen dabei den Prozessorknoten. Bild 5 gibt den "minimal cut" für das obige Beispiel (Bild 4) an.

Der von Ford und Fulkerson entwickelte Netzfluß-Algorithmus ist nur darauf ausgelegt, den Graphen in zwei Teilgraphen zu zerlegen, also nur eine Quelle und eine Senke zu behandeln, was zwei Prozessoren entspricht. Die verschiedenen in der Literatur vorgestellten Verfahren beschäftigen sich nun mit einer Verbesserung dieses Algorithmusses für die Anwendung im Zwei-Prozessor-Fall einerseits und für eine Erweiterung auf mehr Prozessoren andererseits.

Stone stellt eine Erweiterung des beschriebenen Verfahrens für mehrere Prozessoren vor/Sto77a/. Er zeigt, daß das n -Prozessor-Problem auf mehrere Zwei-Prozessor-Fluß-Probleme reduziert werden kann. In jedem Schritt wird der minimale Fluß zwischen zwei Mengen von Prozessoren berechnet. Aus der Zuordnung der Programmknöten zu den Prozessoren in den verschiedenen so erhaltenen Schnittmengen kann man auf die optimale Zuordnung bei n Teilmengen schließen. Das Theorem, auf dem der Stone'sche Algorithmus beruht, führt aber nicht in allen Fällen sofort zu einer eindeutigen Prozeß-Prozessor-Zuordnung. Er selbst vermutet, daß mindestens n^2 verschiedene Schnitte betrachtet werden müssen, um die optimalen Teilmengen für n Prozessoren bestimmen zu können. Dies führt zu enormem Rechenaufwand.

In einem weiteren Artikel/Sto78a/ verweist er auf einen Algorithmus, der zur Auffindung des Optimums bei drei

Prozessoren nur drei Schnitte berechnen muß. Jedoch auch für diesen Algorithmus ist eine allgemeine Anwendbarkeit auf den n -Prozessor-Fall wegen des Rechenaufwandes nicht gegeben. Effizient sei ein Graph mit beliebig vielen Prozessoren nur dann zu lösen, wenn eine baumartige Kommunikationsstruktur (Bild 6) vorläge.

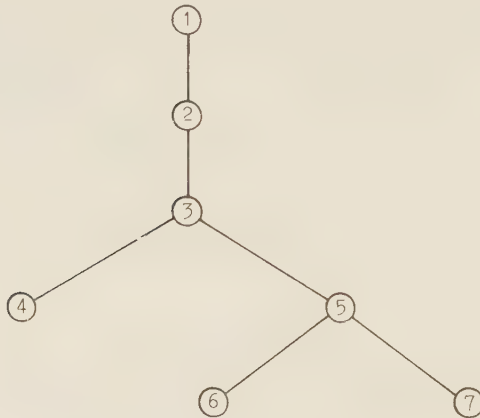


Bild 6: Programmgraph mit baumartiger Kommunikationsstruktur

Der Graph wird in diesem Fall um Anfangs- und Endknoten ergänzt für jedes Ende des Baumes. Für jeden Programmknoten werden n Instanzen generiert, die alle dieselben Eingangs- und Ausgangskanten zur Kennzeichnung des Kommunikationsflusses haben. Der Knoten (i,j) steht jetzt für die Zuordnung des Programms i zu dem Prozessor j . Kanten, die vom Knoten (i,j) zum Knoten (k,j) verlaufen, werden mit den Rechenkosten $r_{k,j}$ des Prozesses k auf dem Prozessor j gewichtet. Führt eine Kante vom Knoten (i,j) zu einem Knoten $(k,1)$, wird sie mit der Summe aus Rechenkosten $r_{k,1}$

des Prozesses k auf Prozessor 1 einerseits und dem wegen des Prozessorwechsels anfallenden Kommunikationsaufwandes $c_{i,k}$ andererseits beschriftet. Gesucht ist jetzt der kostenminimale Fluß vom Anfangsknoten zu den Endknoten, wobei der Pfad immer nur eine Instanz jedes Programmknotens beinhalten darf. Der so erhaltene Pfad kennzeichnet dann die optimale Prozessorzuordnung. Das nächste Bild (Bild 7) zeigt den zum Beispiel aus Bild 6 gehörigen Zuordnungsgraphen, wenn man drei Prozessoren zur Verfügung hat. Die fettgezeichnete Linie könnte solch eine kostenminimale Zuordnung darstellen.

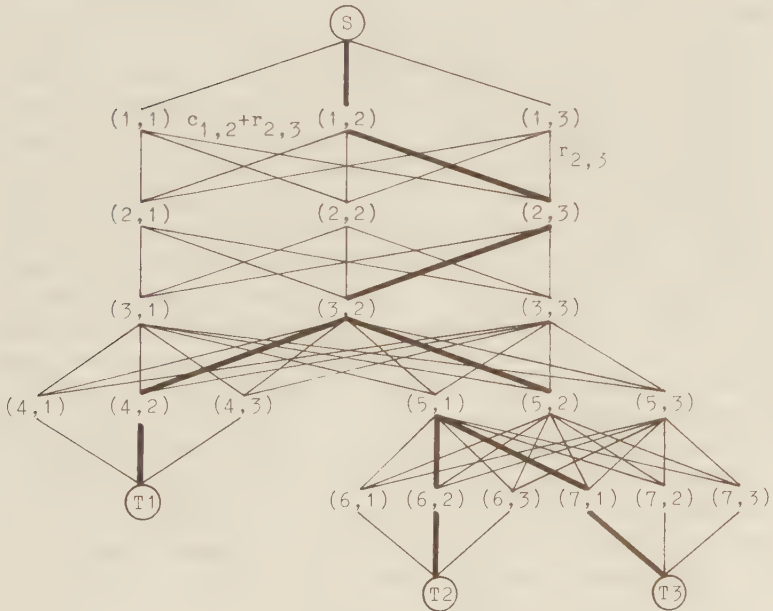


Bild 7: Zuordnungsgraph zu Bild 6

Auch Wu gibt eine Algorithmus an/Wu80a, Wu80b/, der für den Zwei-Prozessor-Fall und für mehr Prozessoren bei baumartiger Kommunikationsstruktur der Programme eine optimale Lösung liefert. Sind mehr als zwei Prozessoren in den Graphen integriert bei beliebiger Verbindungsstruktur der Prozesse, liefert dieses Verfahren nur eine approximative Lösung.

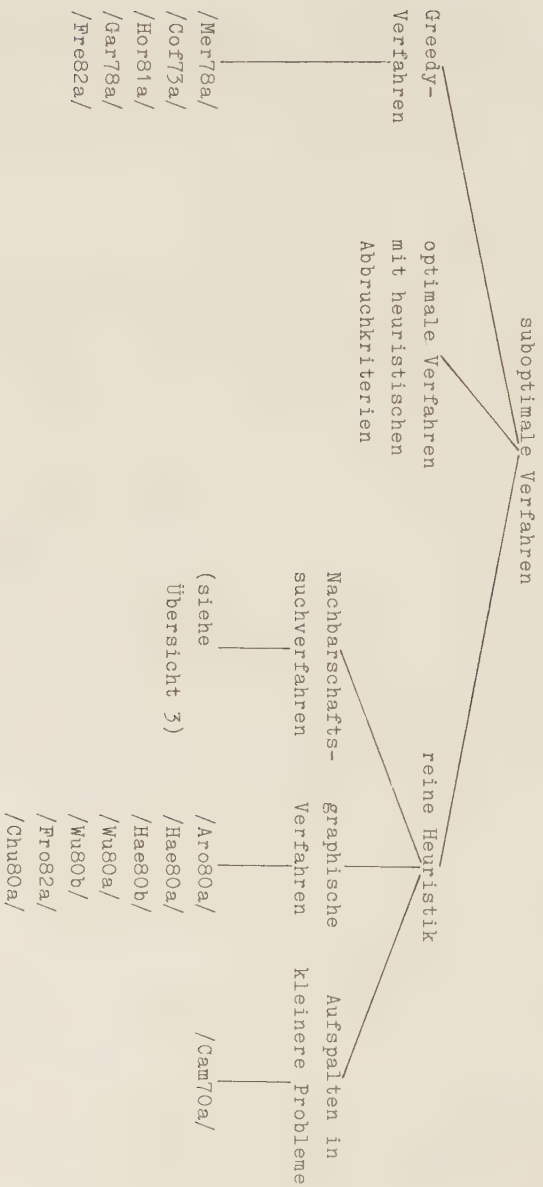
Alle diese Verfahren berücksichtigen jedoch nur den Kommunikationsaufwand und den Gesamtrechnenzeitbedarf. Weitere Restriktionen wie Speicherbeschränkungen, Datenflußrichtungen, Lastausgleich zwischen den Prozessoren und dergleichen werden nicht weiter in die Zuordnung mit einbezogen.

Für den Zwei-Prozessor-Fall hat Stone eine Methode angegeben/Sto78b/, die das Ziel hat, die ungleiche Auslastung der Prozessoren durch einen dynamischen Belastungsausgleich zu verhindern. Für jeden Prozeß wird dabei ein sogenannter "critical load factor" bestimmt, der angibt, wieviel Rechenleistung ein Prozessor dem Prozeß zur Verfügung stellen muß, damit dieser auf dem Prozessor ausgeführt werden kann. Mit dem beschriebenen Netzfluß-Konzept wird zunächst eine optimale Prozessor-Zuordnung gesucht. Während der Laufzeit eines Programms wird dann stets die aktuell zur Verfügung gestellte Rechenleistung des betreffenden Prozessors mit dem "critical load factor" des Prozesses verglichen. Wird eine Überlastung des Prozessors festgestellt, kann eine Prozeßverschiebung zum anderen Prozessor veranlaßt werden. Diese Vorgehensweise setzt jedoch auch wieder voraus, daß jeder Prozessor einen unbeschränkt großen Speicher besitzt, um alle zugewiesenen Prozesse und Daten aufnehmen zu können.

Rao, Stone und Hu stellen daher Methoden vor/Rao79a/, die dieses Problem wenigstens teilweise lösen. Ein Prozessor darf dabei begrenzten Speicher haben, die anderen jedoch müssen immer noch unbeschränkt aufnahmefähig sein. Auch

hier wird zuerst mit der Netzfluß-Technik die kostenminimale Zuordnung ermittelt und anschließend die Speicherbelastung des Prozessors mit dem begrenzten Speicher überprüft. Bei Überlastung werden die Prozesse, die die geringsten Mehrkosten verursachen, auf einen anderen Prozessor verschoben.

Zusammenfassend ist zu den graphentheoretischen Verfahren zu sagen, daß eine Erweiterung der Methoden auf mehr als zwei Prozessoren schwierig oder gar unmöglich ist, zumindest aber daß die Algorithmen mit steigender Anzahl von Prozessoren schnell dermaßen viel Rechenzeit beanspruchen, daß sie für praktische Fälle nicht mehr sinnvoll einsetzbar sind. Desweiteren ziehen die Strategien in den meisten Fällen nur den Kommunikationsaufwand und in einigen Fällen noch den Gesamtrechenbedarf als Optimierungskriterien heran. Andere Restriktionen - wie z. B. Speicherbeschränkungen und gleichmäßige Auslastung der Prozessoren - können entweder gar nicht oder nur mit Schwierigkeiten, meistens irgendwie versteckt in den Kantengewichten, oder erst im nachhinein berücksichtigt werden. Wegen dieser Schwierigkeiten sind diese Methoden nur für einen sehr eingeschränkten Problemkreis sinnvoll anwendbar, sicherlich aber nicht für die hier interessierenden Schedulingprobleme, selbst wenn man die Grenzen noch so eng fassen würde.



Übersicht 2: Suboptimale Verfahren

3.2. Suboptimale Verfahren

Schon während der letzten zwei Jahrzehnte war der Bedarf an brauchbaren Methoden zur Lösung komplexer Probleme zu dringend, um auf mathematisch exakte Optimierungsverfahren zu warten. Deshalb wurden schon immer heuristische Methoden entwickelt, die keine optimale Lösung liefern, aber versuchen, eine einigermaßen gute, zulässige Lösung in vertretbarer Zeit zu finden. Da die Entwicklung solcher heuristischer Methoden mehr eine Kunst als eine Wissenschaft ist, wurde diesen Verfahren jedoch lange wenig Beachtung geschenkt.

Mit der Zeit konnte aber von mehr und mehr allgemein interessanten und in der Praxis vorkommenden Problemkreisen (wie auch von den meisten Scheduling-Problemen) nachgewiesen werden, daß sie zu der Klasse der NP-vollständigen Probleme gehören. Wie schon in Kapitel 2 gezeigt, führt die NP-Vollständigkeit eines Problems dazu, daß man jegliche Hoffnung aufgeben muß, dafür einen Algorithmus zu finden, der eine optimale Lösung liefert und trotzdem immer ein polynomiales Zeitverhalten hat.

In Kapitel 3.1.1. zeigte es sich, daß optimale Verfahren durchaus für bestimmte Aufgabenstellungen ein gutes oder sogar sehr gutes Durchschnittsverhalten haben können, die "worst-case"-Zeitschranke ist jedoch exponentiell. Charlton bemerkt in seinem Artikel/Cha70a/ abschließend, daß solche Algorithmen, die auf einem allgemeinen Ansatz beruhen und somit für einen größeren Problemkreis anwendbar sind, nicht größere und komplexere Probleme lösen können, selbst wenn sie für das gedachte Anwendungsgebiet speziell zugeschnitten sind. Er hält approximative Lösungen für die besten, die man in solchen Fällen erwarten kann.

Gute heuristische Methoden sollten die folgenden drei Eigenschaften erfüllen:

1. Sie sollten Lösungen in vernünftiger Zeit finden.
2. Die Lösungen sollten im Normalfall nahe dem Optimum sein.
3. Die Wahrscheinlichkeit, daß eine einzelne Lösung weit vom Optimum entfernt ist, sollte gering sein.

Bedingung 2 und 3 sind ähnlich, aber nicht gleich. Die Tatsache, daß ein Algorithmus meistens gute Lösungen liefert, schließt nicht aus, daß er in bestimmten Fällen auch einmal nur recht schlechte Zuordnungen findet.

Wie gut diese Anforderungen von einem bestimmten heuristischen Verfahren erfüllt werden, läßt sich im allgemeinen nur durch numerische Tests nachgewiesen. Jedoch kann man durchaus verschiedene Arten heuristischer Methoden voneinander unterscheiden, die jeweils bestimmte Eigenschaften besitzen. Die Kenntnis dieser Grundarten ist wichtig für die Entwicklung heuristischer Methoden für welches Problem auch immer.

Wie findet man nun einen guten Algorithmus, der eine Lösung nahe dem Optimums liefert? Es ist zu erwarten, daß zur Lösung NP-vollständiger Probleme mehr und mehr heuristische Verfahren herangezogen werden, da diese die Struktur des Problems in größerem Maße berücksichtigen können als die allgemeineren Methoden. Sie sind daher zwar nur für einen spezielleren Problemkreis einsetzbar, dafür aber dort um so effizienter. Auf der mit diesen Verfahren ermittelten Lösung könnte aber beispielsweise noch ein "Branch and Bound"-Algorithmus aufsetzen, um das Optimum zu ermitteln. In diesem Fall kann er den Zielfunktionswert dieser ersten Lösung gleich als Schranke verwenden und ist dadurch in der Lage, frühzeitig viele Äste des Lösungsbaums abzuschneiden, weil dieser Wert schon recht nahe am Optimum liegt.

Eine andere Möglichkeit ist es, zwar exakte, optimale Verfahren zu verwenden, aber von vornherein eine gewisse (prozentuale oder absolute) Abweichung vom optimalen Zielfunktionswert einzukalkulieren und den Algorithmus terminieren zu lassen, wenn er eine Lösung gefunden hat, die innerhalb dieses festgelegten Bereichs liegt. Voraussetzung für diese Art der Approximation ist es, daß man den Optimalwert kennt oder zumindest gut abschätzen kann.

Man kann wohl die Klasse der suboptimalen Verfahren ganz grob in drei Gruppen unterteilen (Übersicht 2), die auch schon aus dem eben gesagten ersichtlich sind. Einerseits gibt es eine Vielzahl von "Greedy"-Methoden, die extrem schnell zu einer brauchbaren Lösung führen, falls es überhaupt eine zulässige gibt. Andererseits existieren eine große Anzahl optimaler Verfahren, für die sich heuristische Abbruchkriterien formulieren lassen. Wem diese Arten der Approximation für sein Problem nicht sinnvoll erscheinen, der hat noch die Möglichkeit, ein ganz speziell auf die ihm vorliegende Problematik zugeschnittenes, heuristisches Verfahren auszuarbeiten, entweder durch Entwicklung eines völlig neuen Algorithmusses oder durch geeignete Kombination bekannter Verfahren optimaler oder heuristischer Natur.

Ein heuristisches Lösungsverfahren ist im allgemeinen in zwei Phasen gegliedert:

1. das Eröffnungsverfahren,
2. das suboptimierende Iterationsverfahren.

Mit den Eröffnungsverfahren wird eine erste Ausgangslösung gesucht. Sie bilden die heuristischen Verfahren im engeren Sinne. Die suboptimierenden Iterationsverfahren der zweiten Phase werden zur schrittweisen Verbesserung einer bekannten Ausgangslösung eingesetzt. Das Finden eines Approximationsalgorithmusses für eine bestimmte Problemstellung ergibt sich somit als eine geeignete Kombination

geeigneter Eröffnungsverfahren mit geeigneten suboptimierenden Iterationsverfahren.

Das Zusammenspiel zwischen den Algorithmen der beiden Phasen des Verfahrens kann nun unterschiedlich organisiert werden. Die meisten Eröffnungsverfahren erfordern nur einen geringen Rechenaufwand. Somit lohnt sich häufig der Einsatz verschiedener Verfahren, um aus den mehreren so gewonnenen Ausgangslösungen die beste als Startlösung des Iterationsprozesses auswählen zu können. Viele Eröffnungsverfahren erlauben auch, daß sie selbst von verschiedenen Ausgangspunkten starten, und erzeugen so mehrere unterschiedliche Ausgangslösungen. In der zweiten Phase kann auch der Einsatz mehrerer suboptimierender Iterationsverfahren sinnvoll sein. Man kann verschiedenartige Verfahren auf der selben in der ersten Phase gefundenen Ausgangslösung aufsetzen lassen und dann aus den nicht immer gleichen, verfahrensindividuellen Suboptima das beste als Endlösung auswählen. Aber auch die Hintereinanderschaltung mehrerer verschiedener Iterationsverfahren kann empfehlenswert sein, da häufig das von einem Algorithmus berechnete Suboptimum noch durch eine andere Methode verbessert werden kann.

Bei gewissen Problemen ist es hingegen sinnvoll, das heuristische Verfahren auf die erste Phase zu beschränken, und den Iterationsprozeß ganz wegzulassen. Dies ist dann der Fall, wenn der Rechenaufwand der suboptimierenden Iterationsverfahren vertretbare Grenzen übersteigt oder die mit den Eröffnungsverfahren erzielte Näherungslösung schon so gut ist, daß ein suboptimierendes Iterationsverfahren keine nennenswerte Verbesserung bringen würde. Selbstverständlich hat man auch die Möglichkeit, die Eröffnungsverfahren als Vorstufe eines exakten Optimierungsverfahrens zu verwenden.

3.2.1. "Greedy"-Methoden

Die sogenannten "Greedy"-Methoden werden häufig als Eröffnungsverfahren verwendet. Sie zeichnen sich dadurch aus, daß nur ein einziger Pfad im Lösungsbaum betrachtet wird. Zu diesen Verfahren gehören unter anderem auch die lokalen Planungsregeln aus der Betriebswirtschaft (siehe z. B. Mertens/Mer78a/, Seite 210) oder die Strategien und Prioritätsregeln, nach denen eine Job-Zuteilung im Rechenzentrumsbetrieb bzw. das Scheduling von Programmen auf einer Rechenanlage vorgenommen wird.

Die Vorgehensweise ist recht "straight forward". Aufgrund eines gegebenen Kriteriums wird der jeweils günstigste Nachfolger des gerade aktuellen Knotens im Baum ermittelt. Die einmal getroffene Entscheidung wird nie wieder rückgängig gemacht. Insoweit ähnelt dies Vorgehensweise der dynamischen Programmierung. Dort wurden jedoch durchaus parallel mehrere Pfade betrachtet, falls das Optimierungskriterium dies sinnvoll erscheinen ließ. Bei den "Greedy"-Methoden beschränkt man sich jedoch auf einen einzigen Pfad, wodurch diese Algorithmen sehr schnell sind.

Wohl die meisten der in der Literatur zitierten "Greedy"-Verfahren für den in dieser Arbeit interessierenden Fall des Scheduling in einem Multiprozessorsystem zählen zu der Gruppe der Listenalgorithmen. Sie gehören aufgrund ihrer Einfachheit und Vielfältigkeit zu den bedeutendsten heuristischen Verfahren. Jedes Programm des Aufgabensystems erhält dabei eine Prioritätskennzahl, mit der eine zusätzliche Bedingung dafür angegeben wird, in welcher Reihenfolge die Programme zu bearbeiten sind. Immer wenn ein Prozessor frei wird, wird ihm aus der Menge aller startbereiten Programme das mit der höchsten Priorität zugeordnet. Dabei können die Programme von außen her mit solch einer Priorität versehen werden. Bei Näherungsverfahren ist es jedoch üblich, diese Prioritätskennzahlen in Abhängigkeit von den im System auftretenden Größen zu berechnen, z. B. in Abhän-

gigkeit von Rechenzeit- und Speicherplatzbedarf des Programms oder anderen Betriebsmittelanforderungen. Der Algorithmus ordnet nun den einzelnen vorliegenden Aufgaben Prioritäten zu, wobei die daraus resultierenden Schedules hinsichtlich eines vorgegebenen Kriteriums nicht zu schlecht sein sollen. Aufgrund der dadurch erstellten Prioritätslisten heißen diese Verfahren auch Listenalgorithmen.

Hier sollen vor allem beispielhaft einige der sogenannten statischen Listenalgorithmen betrachtet werden, bei denen die Prioritäten einmal vor der Bestimmung des eigentlichen Schedules ermittelt werden und dann unverändert bleiben. Hingegen kann sich die durch dynamische Listenalgorithmen bestimmte Priorität noch bei Festlegung des Schedules ändern (wohl bemerkt: nur die Priorität der schon bekannten Programme, nicht die Anzahl der zu betrachtenden Aufgaben wie bei dynamischen Scheduling-Problemen).

Der LPTF-Algorithmus ("largest processing time first"), auch kurz LPT ("largest processing time") oder LTF ("largest time first") genannt, ordnet jedem Programm seine Bearbeitungszeit als Priorität zu. Dieser Algorithmus minimiert die maximale Beendigungszeit der Prozessoren und ist besonders gut in Systemen mit voneinander unabhängigen Programmen anwendbar. Er kann so implementiert werden, daß er bei n Prozessoren und m Programmen eine Zeitkomplexität der Ordnung $O(n \cdot \log n)$ hat/Cof73a, Hor81a/. Als Schranke für die Abweichung des mittels LPTF gefundenen Schedules hat sich ergeben, daß für jede beliebige aktuelle Problemstellung I gilt:/Gar78a/

$$\text{LPTF}(I) \leq (4/3 - 1/3m) \text{OPT}(I),$$

wobei OPT ein Algorithmus zur Berechnung des optimalen Schedules ist und $\text{LPTF}(I)$ bzw. $\text{OPT}(I)$ die Länge des mit dem jeweiligen Algorithmus ermittelten Schedules.

Eine Alternative ist es, dem Programm mit der kürzesten Bearbeitungszeit die höchste Priorität zu geben, wie dies

bei dem Algorithmus SPTF ("shortest processing time first") auch namens SPT ("shortest processing time") und STF ("shortest time first"), üblich ist. Der LPTF-Algorithmus minimiert zwar die maximale Beendigungszeit der Prozessoren, besitzt aber die Tendenz, deren mittlere Beendigungszeit zu maximieren. Im Gegensatz dazu minimiert SPTF diese mittlere Beendigungszeit der Prozessoren.

Wieder eine andere Möglichkeit ist es, die Nachfolgerrelation mit in die Prioritätsermittlung einzubeziehen. Bei dem Nachfolger-Algorithmus (N-Algorithmus) bekommt dasjenige Programm die höchste Priorität, das die meisten Nachfolger hat. Durch dieses Verfahren wird berücksichtigt, daß nach dem Start eines Programms mit vielen Nachfolgern im allgemeinen noch verhältnismäßig viel Zeit bis zur Beendigung des gesamten Systems vergeht. Es scheint daher zweckmäßig, Programme mit einer großen Zahl von Nachfolgern so früh wie möglich zu bearbeiten.

Wie der N-Algorithmus vergibt auch der UN-Algorithmus (Unmittelbarer Nachfolger-Algorithmus) die Prioritäten anhand der Nachfolger einer Aufgabe im Bearbeitungsgraphen. Jedoch ergibt sich die Priorität eines Programms hier aus der Anzahl seiner unmittelbaren Nachfolger, die jeweils mit ihrer Bearbeitungszeit gewichtet werden. Der Sinn für eine solche Vergabe ist darin zu sehen, daß nach dem Start eines Programms mit vielen unmittelbaren Nachfolgern diese möglicherweise nach dessen Beendigung selbst gestartet werden können. Es ist zu erwarten, daß, je größer die Anzahl der unmittelbaren Nachfolger ist, desto eher sind alle Prozessoren bei der nächsten Zuordnung eines Programms zu einem Prozessor - aufgrund der größeren Anzahl der dann zur Bearbeitung anstehenden Programme - aktiv.

Nicht auf der Rechenzeit der einzelnen Programme, sondern auf deren Speicherplatzbedarf basiert die Prioritätsvergabe des LMF-Algorithmus ("largest memory first"). Das Programm mit der größten Speicherplatzanforderung erhält die höchste Priorität. Diese Vergabestrategie kann sich

besonders beim Scheduling speicherresidenter Programme als nützlich erweisen, da es dort andernfalls vorkommen kann, daß alle Prozessoren so mit Programmen mit niedrigerem Speicherplatzbedarf belegt sind, daß die "großen Brocken" nirgendwo mehr hineinpassen und so die gesamte Zuteilung mittels eines "Greedy"-Verfahrens anderer Art als LMF unmöglich wird.

Auch eine Kombination der beiden Strategien LPTF und LMF ist denkbar: In erster Linie bekommen Programme mit höherem Speicherbedarf höhere Prioritäten. Haben mehrere Programme jedoch innerhalb gewisser Grenzen gleichen Speicherbedarf, werden sie anhand ihrer Bearbeitungszeit geordnet. Diesen Algorithmus nennt man LMTF ("largest memory / time first").

Wieder anders kombiniert der LPF-Algorithmus ("largest product first") die beiden Ordnungskriterien der LPTF- und LMF-Algorithmen. Er bildet für jeden Prozeß das Produkt aus Speicherplatzbedarf und Bearbeitungszeit und ordnet sie anhand dieses Produkts.

Es wurden jetzt eine ganze Reihe solcher heuristischer Regeln zur schnellen Ermittlung eines guten, approximativen Schedules vorgestellt. French/Fre82a/ gibt noch eine Reihe anderer Prioritäts- und Auswahlregeln an. In dieser Art lassen sich überdies viele andere konstruieren, dem jeweiligen Problemkreis angepaßt.

3.2.2. Exakte Verfahren mit heuristischem Abbruchkriterium

Es wurde gezeigt, daß trotz der beachtlichen Verfeinerung der Entscheidungsbaumverfahren und der Leistungssteigerung moderner Rechenanlagen der sinnvollen Anwendung exakter Verfahren immer noch ein in der Regel nicht zu vertretender Rechenaufwand entgegensteht. Dieser Aufwand

rührt vor allem daher, daß in dem im Verlauf des Lösungsprozesses entstehenden Entscheidungsbaum zahlreiche Äste abgearbeitet werden müssen, die ohne Nutzen für die eigentliche Bestimmung des Optimums sind. Je früher man die Nutzlosigkeit eines solchen Zweiges erkennt, desto geringer ist der Rechenaufwand des Verfahrens. Es liegt daher nahe, Zweige des Lösungsbaumes schon dann abzuschneiden, wenn man ihre Nutzlosigkeit nur vermutet. Andererseits birgt dieses Abtrennen von Zweigen des Entscheidungsbaumes aufgrund nur vermuteter Nutzlosigkeit die Gefahr in sich, daß dabei auch besonders gute oder sogar die optimale Lösung abgeschnitten werden.

Man muß also sinnvolle und plausible Kriterien für ein solches Kappen von Entscheidungsbäumen finden. Sie lassen sich praktisch nur unter Heranziehung problemspezifischer Eigenschaften entwickeln. Entsprechende heuristische Kriterien beziehen sich unter anderem auf die Beschränkung der Rechenzeit, auf die Festsetzung von Obergrenzen für die Zahl der zu entwickelnden Knoten im gesamten Baum bzw. innerhalb der einzelnen Zweige, auf die Beschränkung der insgesamt zu untersuchenden Zweige oder auf die Verschärfung einer unteren Grenze zum Verwerfen von Teillösungsmengen.

Zur Begründung für diese und ähnliche heuristische Kriterien zum vorzeitigen Abschneiden von Zweigen sei noch einmal auf das verwiesen, was schon bei der Beschreibung der "Branch and Bound"-Verfahren erwähnt wurde. Jene finden in vernünftiger Zeit schon eine gute oder gar optimale Lösung, lange bevor sie nachweisen können, daß diese optimal ist. Diese Eigenschaft kann deshalb zur Konstruktion eines approximativen Algorithmusses ausgenutzt werden.

Eine Möglichkeit ist es daher, den Algorithmus terminieren zu lassen, wenn er eine Lösung gefunden hat, die nahe dem Optimum liegt. Die Nähe kann dabei entweder absolut oder als prozentuale Abweichung formuliert sein. Mit diesem Abweichungswert hat man gleichzeitig eine Angabe über den größtmöglichen Fehler, der bei der so gewonnenen Lösung

auftreten kann. Der Rechenzeitaufwand wird deutlich eingeschränkt, ist aber trotzdem nicht völlig abschätzbar, da es immer noch von den Werten der aktuellen Problemstellung abhängt, wie schnell der Zielfunktionswert der enumerierten Lösungen gegen das Optimum konvergiert.

Um diese Art der Approximation anwenden zu können, ist eine genaue Kenntnis des zu erwartenden Optimalwertes nötig. Zumindest aber muß man ihn einigermaßen genau abschätzen können. Dies wird nicht bei allen Problemkreisen und allen Optimierungskriterien möglich sein.

Eine andere Art, ein exaktes Verfahren in der Rechenzeit zu beschränken, ist es - wie schon erwähnt -, eine bestimmte Gesamtzahl von Iterationen (ermittelten Lösungen) festzulegen. Die letzte Lösung vor Abbruch des Verfahrens ist die bis dahin beste gefundene. Mit dieser Art der Approximation kann man relativ genaue Aussagen darüber machen, wie lange der Algorithmus brauchen wird. Hingegen kann man nicht abschätzen, in welcher Größenordnung sich die Abweichung vom Optimum bewegen wird. Hier braucht es viel Feingefühl, die richtige Anzahl an Iterationen vorzugeben, um einerseits nicht übermäßig viel Rechenzeit zu benötigen, andererseits aber doch eine gewisse gewünschte Nähe zum Optimum nicht zu verlieren.

Der Fantasie bei der Entwicklung verschiedenster heuristischer Abbruchkriterien sind keine Grenzen gesetzt. Sie lassen sich gut für sehr eingegrenzte Problemkreise finden, indem man viel Kenntnis über die innere Struktur des angesprochenen Kreises einfließen läßt.

3.2.3. Rein heuristische Verfahren

Für viele Probleme sind beide eben geschilderten Arten der Approximation unbefriedigend. Die "Greedy"-Verfahren sind einerseits sehr schnell, liefern aber andererseits häufig Lösungen, die zwar schon recht gut, aber noch relativ

weit vom Optimum entfernt sein können. Bei der anderen beschriebenen Art von Näherungsverfahren lassen sich dagegen nicht immer erfolgreich gute Kriterien zum Abbruch optimaler Methoden finden, bzw. manchmal sind diese Verfahren überhaupt nicht anwendbar. In solchen Fällen empfiehlt es sich, ein Verfahren zu entwickeln, das speziell auf das vorliegende Problem zugeschnitten ist. Dabei können durchaus Elemente der bisher geschilderten Algorithmen in diesen Ansätzen wieder auftauchen.

Für die so entstehenden doch recht problemspezifischen Verfahren läßt sich keine generelle Methodik darstellen. Die Vorgehensweise dabei bedarf viel Fingerspitzengefühls, vieler Tests und viel Erfahrung. Deshalb seien hier exemplarisch nur kurz noch einige auf Graphen arbeitende Verfahren dargestellt, sowie ein Beispiel dafür, wie ein komplexeres Problem zwecks der Approximation auf mehrere einfache zurückgeführt wird.

Eine andere Gruppe recht leistungsfähiger, heuristischer Verfahren sind die Tauschverfahren, mit denen sich das nächste Kapitel näher beschäftigt.

3.2.3.1. Graphentheoretische und Clustering-Verfahren

Schon bei den exakten Verfahren wurden einige Algorithmen beschrieben, die auf graphentheoretischen Methoden beruhen. Solch ein Ansatz läßt sich selbstverständlich auch für die Entwicklung heuristischer Verfahren verwenden.

Arora und Rana/Aro80a/ schlagen einen Algorithmus vor, in dem bei jedem Schritt ein beliebiger Prozeßknoten des Graphen ausgewählt und mit dem Prozessorknoten verschmolzen wird, zu dem die Kante mit dem größten Gewicht führt. Anschließend muß noch der Graph entsprechend der vorgenommenen Verschmelzung modifiziert und die Gewichte der vom Verschmelzungsknoten ausgehenden Kanten korrigiert werden. Bei n Prozeß- und m Prozessorknoten muß man insgesamt nur m

Verschmelzungen durchführen. Maximal können im Graphen $n+m-1$ Kanten vorkommen, die korrigiert werden müssen. Somit besitzt dieser Algorithmus eine Zeitkomplexität der Ordnung $O(n \cdot m + m^2)$.

Die so gefundene erste zulässige Zuordnung kann nun bei Bedarf weiter verbessert werden, indem man sich einen Prozeß herauspicks und nachsieht, ob sich dieser nicht auf einen anderen Prozessor verschieben läßt, so daß der Gesamtaufwand geringer wird. Das kann so lange fortgeführt werden, bis sich kein Prozeß mehr verschieben und sich folglich die gefundene Zuordnung nicht mehr verbessern läßt.

Ähnlich arbeitet auch der von Haessig und Jenny vorgeschlagene Algorithmus/Hae80a, Hae80b/. Zum Finden der ersten Lösung wird hier jedoch nicht von beliebigen Prozeßknoten ausgegangen. Die Kanten zwischen den Knoten werden zuerst in absteigender Reihenfolge ihrer Gewichte geordnet und in dieser Reihenfolge überprüft, ob es sinnvoll ist, sie zu eliminieren.

Hat man so eine erste Zerlegung des Graphen erhalten (Bild 8), bei der jeder Teilgraph nur einen Prozessorknoten besitzt, wird in der zweiten Phase des Algorithmusses (Bild 9) versucht, diese Aufteilung durch Verschieben von Prozeßknoten zu einem anderen Prozessor zu verbessern. Die zu verschiebenden Knoten werden durch das bereits in Kapitel 3.1.3. genannte Max-Flow/Min-Cut-Theorem von Ford und Fulkerson/For62a/ ermittelt. Es werden solange Knoten verschoben, bis entweder keine Verbesserung mehr erreicht werden kann oder der dafür benötigte Aufwand unrentabel wird.

Haessig und Jenny weisen darauf hin, daß dieser Algorithmus nur auf Zuweisungsprobleme ohne jegliche Beschränkungen bezüglich Speicherplatz, Auslastung und dergleichen angewendet werden kann. Eine Erweiterung des Algorithmus um eine dritte Phase könne aber solche Probleme lösbar machen.

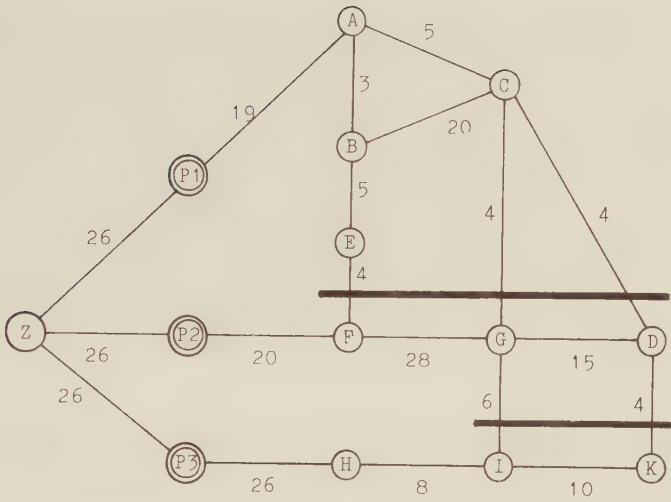


Bild 8: Programmgraph mit Ausgangslösung (Phase 1)

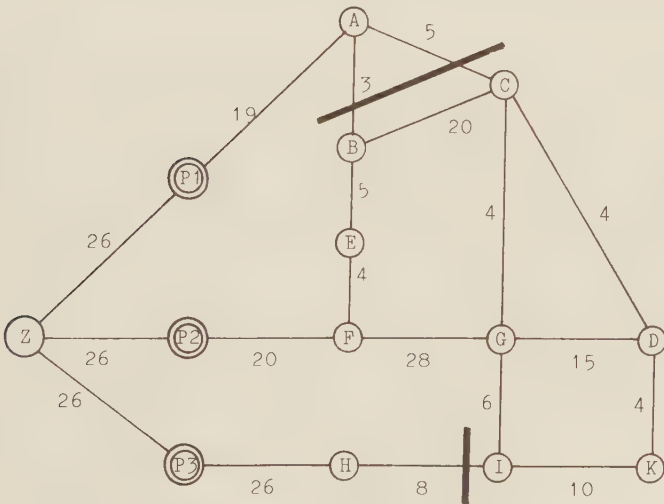


Bild 9: Programmgraph mit Endlösung (Phase 2)

Wie ebenfalls in Kapitel 3.1.3. erwähnt, geben auch Wu und Liu/Wu80a, Wu80b/ einen Algorithmus an, der im allgemeinen Fall eine suboptimale Lösung liefert, Im Zwei-Prozessor-Fall und beim Vorliegen einer baumartigen Kommunikationsstruktur hingegen sogar eine optimale. Der Algorithmus beruht auch darauf, den gesamten Graphen so zu zerlegen, daß jeder der Teilgraphen genau einen Prozessorknoten beinhaltet. Eine darauf aufbauende zweite Phase zur Verbesserung wird hier nicht vorgeschlagen. Damit ergibt sich eine Komplexität des Verfahrens von $O(n)$ bei n Prozessoren, die gleich der der ersten Phase des Algorithmusses von Arora und Rana ist.

Der überwiegende Nachteil all dieser mit graphentheoretischen Methoden arbeitenden Verfahren ist, daß sie meist nur die Kommunikationsminimierung zum Ziele haben und Restriktionen entweder überhaupt nicht oder nur schwer und im nachhinein berücksichtigt werden können. Dasselbe gilt für eine gleichmäßige Auslastung der Prozessoren.

Ein anderer Ansatz zum Lösen von Zuordnungsproblemen beruht auf Clustering-Verfahren. Sie bilden eine Klasse recht uneinheitlicher Methoden, die teils mehr statistisch, teils mehr graphentheoretisch orientiert sind. Um Clustering-Methoden anwenden zu können, muß zunächst die graphische Darstellung des Problems in eine Punktmengen-Darstellung im mehrdimensionalen Raum transformiert werden. Die Punkte entsprechen dabei den Knoten im Graphen. Ihr Abstand voneinander gibt den Grad ihrer Zusammengehörigkeit an. Z. B. kann bei Programmen ein hohes Kommunikationsaufkommen als besonders große Zusammengehörigkeit interpretiert werden. Ein besonders kritischer Punkt bei allen Clustering-Verfahren ist die Wahl eines geeigneten Abstandsmaßes, für die es viel Feingefühls bedarf, um das für das anstehende Problem beste auszusuchen. Aufgabe des Clustering-Algorithmusses ist es dann, in dieser Punktmenge mit Hilfe des

Abstandsmaes besonders dichte Gruppierungen ("Cluster") aufzufinden. Zurcktransformiert in einen Graphen bilden sie die Elemente der gesuchten Zerlegung.

Eine bersicht ber die verschiedensten Abstandsmae und Clustering-Verfahren kann man bei Fromm/Fro82a/ finden. Wie bei den graphentheoretischen Methoden zeigen aber auch die Clustering-Verfahren den Nachteil, da man nur schwer andere Kriterien als die Kommunikationsminimierung in den Algorithmus mit einbeziehen kann. Sie knnen hier jedoch bedingt als Nebenbedingung einflieen.

Chu et. al. schlagen vor/Chu80a/, ein kommunikationsminimierendes Clustering-Verfahren zur Ermittlung einer ersten guten Lsung einzusetzen und darauf aufbauend andere Anforderungen, wie gleichmige Auslastung der Prozessoren oder Speicherbeschrnkungen, in einer zweiten Phase zu berprfen und dabei gegebenenfalls einzelne Prozesse noch zu verschieben.

3.2.3.2. Aufspaltung des Problems

Fr etliche Scheduling-Probleme gibt es effiziente Algorithmen, sofern das Programmsystem nicht auf mehr als zwei Prozessoren verteilt werden soll. Diese Tatsache machen sich Campbell, Dudek und Smith/Cam70a/ fr das von ihnen vorgeschlagene heuristische Verfahren zur Lsung von Scheduling-Problemen zunutze.

Sie bilden aus dem gesamten Problem von n Programmen und m Prozessoren eine beliebige Anzahl von kleineren Problemen mit n Programmen und 2 Maschinen. Diese Teilprobleme werden unabhngig voneinander gelst und anschlieend die so erhaltenen optimalen Teillsungen zu einer suboptimalen Gesamtlsung verschmolzen. Mit dieser Vorgehensweise lt sich oft sogar die optimale Lsung des ursprnglichen Problems finden.

Der Rechenaufwand für praktisch interessante Probleme ist jedoch immer noch beträchtlich aufgrund der Vielzahl kleinerer, optimal (wenn auch effizient) zu lösender Teilprobleme.

4. Nachbarschaftssuchverfahren

Bisher wurde gezeigt, daß bei der exakten, optimalen Lösung eines Problems, welches die Eigenschaft der NP-Vollständigkeit besitzt, wie z. B. einem Scheduling-Problem, ein in der Regel unvertretbar hoher Rechenaufwand entgegensteht. Aber auch eine Approximation solcher exakter Verfahren oder deren Begrenzung durch heuristische Schranken oder Abbruchkriterien führt immer noch zu Algorithmen, die rechen-technisch zu aufwendig sind, um in der Praxis wirklich interessant zu sein. Welche Möglichkeiten bieten sich noch an?

Eine Weg, eine gute Lösung für ein Problem zu finden, ist, einen iterativen Suchprozeß zu starten, in dem unter Berücksichtigung der Struktur der Nebenbedingungen und der Zielfunktion versucht wird, aus einer gegebenen Lösung jeweils direkt eine neue verbesserte Lösung zu bestimmen.

4.1. Grundlegende Prinzipien

Die Vorgehensweise bei einem solchen Verfahren ist die, mit einer zulässigen Lösung zu beginnen und von ihr aus eine Folge unmittelbar aufeinander aufbauender, zulässiger Lösungen mit monoton steigendem bzw. bei Minimierungsproblemen fallendem Zielfunktionswert zu bilden. Jede neue Bezugslösung (Nachfolgerlösung) dieser Folge wird unter Zuhilfenahme bestimmter Auswahl- und Entscheidungskriterien innerhalb einer noch näher zu beschreibenden, überschaubaren und algorithmisch schnell abzusuchenden Umgebung oder Nachbarschaft der Bezugslösung gesucht. Dieser Prozeß wird so lange fortgesetzt, bis man eine Lösung gefunden hat, in deren Nachbarschaft keine bessere enthalten ist. Damit hat man ein verfahrensspezifisches lokales Optimum erreicht, welches von der Anfangslösung, der Struktur der Nachbarschaften und den jeweils herangezogenen Auswahl- und Entscheidungskriterien abhängt.

Methoden, die auf solch einem iterativen, streng vorwärts orientierten Suchprozeß basieren, bei dem stets ein unmittelbarer Zusammenhang zwischen den einzelnen Bezugslösungen besteht, werden als Nachbarschaftssuchverfahren bezeichnet.

Ein solches Suchverfahren setzt eine heuristische Strategie voraus, die geeignet ist,

1. eine erste (Ausgangs-)Lösung zur Verfügung zu stellen,
2. auf der Basis einer Bezugslösung die in der Umgebung oder Nachbarschaft liegenden Lösungskandidaten zu ermitteln,
3. für die Auswahl bzw. Beurteilung der Kandidaten geeignete Kriterien bereitzustellen und
4. die Endlichkeit des gesamten Suchprozesses zu garantieren.

Die in Punkt 4 geforderte Garantie der Endlichkeit des Suchprozesses ist durch die Forderung nach einem monotonen Anstieg bzw. Abfall des Zielfunktionswertes gegeben, da der Lösungsbereich insgesamt als abgeschlossen angesehen wird. Die einzelnen in der Literatur vorgeschlagenen Verfahren, die sich der skizzierten Vorgehensweise bedienen, unterscheiden sich deshalb lediglich in der individuellen Realisierung der ersten drei Anforderungen. Der generelle Ablauf hingegen ist bei allen gleich.

4.1.1. Die Ausgangslösung

Wie wählt man nun eine erste Lösung? Diese Wahl hängt - ebenso wie die Wahl des Verfahrens zur Ermittlung der in der Nachbarschaft liegenden Lösungskandidaten - sehr von der Problemklasse ab, für die eine Lösung gesucht wird.

Eine Einsatzmöglichkeit der Nachbarschaftssuchverfahren ist, zu einer optimalen, aber nicht ganzzahligen Lösung eine gute ganzzahlige zu finden. Für solche Probleme wird häufig als erstes ein allgemeines LP-Verfahren eingesetzt, daß die optimale, nicht ganzzahlige Lösung als Ausgangspunkt für den nachfolgenden Suchprozeß ermittelt. Aber es sind auch Verfahren denkbar, die direkt eine der ganzzahligen Lösungen, die in der unmittelbaren Nähe der nicht ganzzahligen liegen, als erste Bezugslösung ermitteln.

Ein anderer Ansatz, eine erste nicht triviale zulässige Lösung zu generieren, ist der Einsatz eines geeigneten "Greedy"-Verfahrens (siehe Kapitel 3.2.1.). Andere, dem jeweiligen Problemkreis angepaßte Methoden sind denkbar.

4.1.2. Die Nachbarschaft

Um die Nachbarschaftssuchverfahren recht zu verstehen, muß noch das Konzept der Nachbarschaft etwas näher ansehen werden. Wie schon die Wahl der Ausgangslösung ist auch dieses Konzept direkt von dem Problemkreis abhängig, für den eine Lösung gesucht wird.

Es wird eine sogenannte Nachbarschaftsstruktur/Rei65a, Gal76a/ eingeführt. Dies ist eine allgemeine strukturelle Beschreibung aller relevanten Nachbarschaftsbeziehungen zwischen den einzelnen Lösungen des Problems. Sie kann als ein ungerichteter Graph angesehen werden, dessen Knoten jeweils eine Lösung darstellen und dessen Kanten kennzeichnen, daß die durch sie miteinander verbundenen Lösungen in einem bestimmten Sinne benachbart sind.

Sucht man z. B. zu einem allgemeinen Problem der linearen Programmierung eine ganzzahlige Lösung, so entsprechen die Knoten des Graphen den ganzzahligen Lösungen. Die typische Gitteranordnung der ganzzahligen Lösungspunkte bietet sich unmittelbar als eine mögliche

Nachbarschaftsstruktur an. Darin sind jeweils nur solche Lösungspunkte benachbart, die sich nur in einer Variablen um den Wert 1 unterscheiden. Darauf lassen sich dann beliebige komplexere Nachbarschaftsstrukturen aufbauen. So kann man indirekte Verbindungen über mehrere Kanten zusammenfassen und so Strukturen erhalten, die die Änderung eines Variablenwertes um einen größeren Betrag als 1 oder die gleichzeitige Änderung mehrerer Variablenwerte um einen Betrag größer oder gleich 1 erfassen.

Wie sieht nun solch eine Nachbarschaftsstruktur bei Scheduling-Problemen aus? Seien dazu zwei Beispiele betrachtet.

Als erstes soll der Fall betrachtet werden, daß für n Programme, die alle auf einem Rechner laufen sollen und jeweils bestimmte Beendigungstermine haben oder eine bestimmte Abarbeitungsreihenfolge einhalten müssen, ein Schedule gefunden werden soll.

Eine Lösung sei gegeben durch die Reihenfolge

$$(P_1, P_2, P_3, P_4, \dots, P_{n-1}, P_n)$$

der Programme. In der Nachbarschaft dieses Schedules gibt es $n-1$ verschiedene Schedules, die durch einfachen Tausch zweier benachbarter Programme entstehen.

$$(P_2, P_1, P_3, P_4, \dots, P_{n-1}, P_n)$$

$$(P_1, P_3, P_2, P_4, \dots, P_{n-1}, P_n)$$

$$(P_1, P_2, P_4, P_3, \dots, P_{n-1}, P_n)$$

.....

$$(P_1, P_2, P_3, P_4, \dots, P_n, P_{n-1})$$

Tauscht man nun alle möglichen Paare von Programmen, erhält man $n \cdot (n-1)/2$ Schedules, die zusammen die gesamte, durch einfachen paarweisen Tausch gewonnene Nachbarschaft eines Schedules bilden. Auf gleiche Weise lassen sich kompliziertere Nachbarschaftsstrukturen bilden, in denen nicht nur der Tausch zweier Programme betrachtet wird.

Als weiteres Beispiel sei noch das einfache Zuordnen mehrerer Prozesse auf mehrere Rechner ohne weitere Beschränkungen betrachtet. Angenommen man habe fünf Prozesse, A, B, C, D und E, und drei Prozessoren P1, P2 und P3. Eine Zuordnung könnte z. B. so aussehen:

	A	B	C	D	E
P1	x	x			
P2			x	x	
P3					x

Ein einfacher Tausch bedeutet in diesem Fall, daß ein Prozeß auf einen anderen Rechner verlagert wird und im Gegenzug ein diesem Prozessor zugeordnetes Programm auf den Rechner des ersten wechselt. In dem obigen Beispiel hätte man die Möglichkeit, entweder Programm A mit Programm C auszutauschen, oder mit D oder E, oder B mit C, D oder E, oder C mit E oder D mit E. Diese Alternativen bilden wieder die Nachbarschaft für die obige Ausgangslösung bezüglich eines einfachen Austausches. Ebenso lassen sich die Alternativen bilden, wenn mehr als zwei Programme in den Tausch einbezogen werden sollen.

Dem aufmerksamen Leser wird es nicht entgangen sein, daß bei den beiden Beispielen von Scheduling-Problemen die Nachbarschaft anders definiert wurde, als in dem Fall der Suche einer ganzzahligen Lösung zu einem LP-Problem.

Während dort eine oder mehrere Variablen in ihrem Wert verändert (erhöht oder erniedrigt) wurden, bleiben beim Scheduling die Werte der Variablen unverändert, nur daß zwei oder mehr Variablen ihre Werte gegeneinander austauschen. Demzufolge lassen sich die Nachbarschaftssuchverfahren einteilen in

1. die Tauschverfahren,

bei denen die Werte von Variablen gegeneinander ausgetauscht werden aber in ihrem Betrag unverändert bleiben,

und

2. die direkten Suchverfahren,

bei denen die Werte einer oder mehrerer Variablen in ihrem Betrag verändert werden.

4.1.3. Auswahl- und Entscheidungskriterien

Um nun aus einer Nachbarschaft eine Nachfolgelösung mit einem höheren (oder zumindest mit keinem niedrigeren) Zielfunktionswert als Ausgangspunkt der nächsten Iteration auszuwählen, bzw. um die lokale Optimalität einer Bezugslösung innerhalb der betreffenden Nachbarschaft nachweisen zu können, sind bestimmte Auswahl- und Entscheidungskriterien notwendig. Reiter und Sherman/Rei65a/ fassen diese Kriterien in einer sogenannten Nachfolgerfunktion zusammen.

Im allgemeinen wird man davon ausgehen, daß sich in der Nachbarschaft einer Bezugslösung lauter Kandidaten befinden, die unterschiedliche Zielfunktionswerte haben und von denen mindestens einer besser ist als der der momentan aktuellen Bezugslösung, soweit diese nicht schon ein lokales Optimum darstellt. Eine dieser Nachbarschaftslösungen soll nun als Nachfolger des augenblicklichen Ausgangspunktes der Iteration ausgewählt und zur neuen Bezugslösung werden.

Hierzu bieten sich verschiedene Möglichkeiten an. Man kann die Lösung mit dem besten Zielfunktionswert wählen oder eine zufällige, nicht am Zielfunktionswert orientierte Auswahl treffen. Auch Prioritätskennziffern lassen sich verwenden, für die sich insbesondere die Auswahlkriterien der Simplexmethode in modifizierter Form eignen. Auch ist es möglich, einfach immer die erste gefundene Lösung der Nachbarschaft, die einen besseren Zielfunktionswert als die Bezugslösung hat, zu wählen und die übrigen in der Nachbarschaft liegenden Lösungen gar nicht explizit zu berechnen. Andere Kriterien lassen sich je nach Bedarf entwickeln.

Die Wahl dieser Kriterien hängt eng mit der Wahl der Nachbarschaftsstruktur zusammen. Hat man sich dort für einen Algorithmus entschieden, der nur wenige Kandidaten zur weiteren Auswahl generiert, kann man jetzt diese alle betrachten und den besten auswählen. Wenn hingegen die Nachbarschaft der momentanen Bezugslösung sehr groß ist, würde es zu viel Zeit beanspruchen, alle Kandidaten zu prüfen. In diesem Fall wird man es vorziehen, sofort nach der Generierung eines Kandidaten nachzusehen, ob diese Lösung zu einer Verbesserung des Zielfunktionswertes führt. Ist das der Fall, wird dieser Kandidat sofort neue Bezugslösung. Auf diese Weise wird die gesamte Nachbarschaft nur äußerst selten vollständig generiert, üblicherweise nur im letzten Zyklus, in dem keine Verbesserung des Zielfunktionswertes mehr ermittelt werden kann.

Der bisher skizzierte Prozeß ist auf die Suche nach einer verbesserten Lösung ausgerichtet. Etwas anders, wenn auch ähnlich, gestaltet er sich, falls dieses Verfahren auch zur Ermittlung einer ersten zulässigen Lösung eingesetzt werden soll. Da in diesem Fall die aktuellen Bezugslösungen noch unzulässig sind, können sich die Auswahl- und Entscheidungskriterien nicht mehr an einer Verbesserung des Zielfunktionswertes orientieren, sondern müssen in erster Linie darauf ausgerichtet sein, die Unzulässigkeit der betrachteten Bezugslösung zu vermindern.

Für Probleme, bei denen eine ganzzahlige Lösung zu einem nicht ganzzahligen Optimum gesucht wird, lassen sich dafür in Anlehnung an die Simplex-Methode Maße ableiten, anhand derer die Unzulässigkeit der Bezugslösung vermindert werden kann. Solche sich an der Unzulässigkeit einer Lösung orientierenden Auswahlkriterien stellen jedoch aufgrund der Suboptimalität dieser Verfahrensart nicht sicher, daß in jedem Fall eine der existierenden zulässigen ganzzahligen Lösungen gefunden wird. Es kann nicht ausgeschlossen werden, daß der Suchprozeß in einer Lösung mit lokal minimaler Unzulässigkeit endet.

Die Bestimmung der zu einer kontinuierlichen Bezugslösung benachbarten ganzzahligen und möglicherweise (aber nicht notwendigerweise) auch zulässigen Lösung kann aber beispielsweise auch mit Hilfe von Rundungsprozeduren erfolgen. Kann auf diesem Weg keine ganzzahlige Lösung in der Nachbarschaft ermittelt werden und steht keine neue kontinuierliche Bezugslösung zur Verfügung, bietet es sich an, auf zufällige Weise oder mit Hilfe funktioneller Beziehungen aus dem kontinuierlichen Lösungsbereich des korrespondierenden LP-Problems weitere Bezugslösungen auszuwählen. Da hierdurch die Endlichkeit des Suchprozesses gefährdet sein kann, müssen besondere Vorkehrungen getroffen werden, um die Terminierung des Verfahrens zu gewährleisten.

4.2. Einige Verfahren

Wie schon erwähnt, ist die prinzipielle Vorgehensweise bei allen Verfahren, die solche Such- und Tauschprozesse beinhalten, dieselbe. Daher bleiben nur drei Fragestellungen offen, zu deren Beantwortung die verschiedenen, auf dem generellen geschilderten Prinzip aufbauenden, heuristischen Verfahren eine Wahl treffen müssen:

1. Wie wählt man eine Anfangslösung?
2. Welches Verfahren wählt man, um eine bestimmte (ebenfalls auszusuchende) Nachbarschaft zu ermitteln?
3. Welche Kriterien zieht man heran, um aus der Menge der ermittelten Nachbarlösungen eine als künftige neue Bezugslösung auszuwählen?

Einen Überblick über die in den nachfolgenden Abschnitten dargestellten Verfahren gibt die Übersicht 3.

Direkte Suchverfahren:

	Erstlösung vorge- geben	aus IP	trivial, unzul.	Nachbarschaft 1 Var. 2 Var.	> 2 Var.	Auswahlkriterium Max.- Gewinn	< max. Gewinn	Fernhalten von Grenzen	Sackgassen- ausbruch
Echols et al. /Ech68a/ Garfinkel et al. /Gar72a/ Hillier /Hil69a/ Müller-Werbach /Mül69a/	x	x	x	x	x	x	x	x	x

Tauschverfahren:

	Erstlösung vorge- geben	aus IP	zufällig	Nachbarschaft 2 Var. 3 Var.	Auswahlkriterium Max.- Gewinn	bestes besserer	bestes aus begrenzter Nachbarsch.	Sackgassen- ausbruch
Bokhari /Bok81a/ Kreuzberger /Kre69a, Kre70a/ Reiter et al. /Rei65a/	x	x	x	x	x	x	x	x

Übersicht 3: Nachbarschaftssuchverfahren

4.2.1. Direkte Suchverfahren

Mit einer vorgegebenen Lösung startet das von Echols und Cooper/Ech68a/ vorgeschlagene Verfahren. Es ist speziell für das Lösen von Integer-Programmierungsproblemen gedacht. Als erstes sollen hier die einzelnen Variablen des Gleichungssystems daraufhin untersucht werden, welche den größten Gewinn für die Erreichung des durch die Zielfunktion gegebenen Zieles erbringen würde. Diese wird dann entsprechend modifiziert und das System daraufhin neu überprüft, bis man eine Lösung erhält, die sich nahe einer durch eine Restriktion gegebenen Grenze befindet.

Um nicht frühzeitig in eine Sackgasse zu geraten, wird dann in der zweiten Phase des Algorithmusses eine Variablen um einen festen Wert verändert. Dabei wählt man dafür die Variable so aus, daß sich der Zielfunktionswert durch diese Veränderung verbessert, jedoch ungeachtet dessen, ob dabei eine zulässige Lösung entsteht oder nicht. Ist die Lösung nicht zulässig, wird, wenn möglich, eine zweite Variable entsprechend verändert. Anschließend wird der Wert jeder Variablen halbiert und mit dieser neuen Lösung wieder die erste Phase gestartet.

Läßt sich in diesem Zyklus keine Verbesserung mehr erreichen, schließt sich eine weitere Phase an, die entsprechend der Phase 2 arbeitet, nur daß sich gleichzeitig immer zwei Variablen ändern. Dies wird wiederholt, bis sich auch hier keine Verbesserung des Zielfunktionswertes mehr erreichen läßt. Diese letzte Phase ist erwartungsgemäß sehr rechenaufwendig. Bei vielen Problemen, in denen in den ersten Phasen eine Lösung nahe des Optimums gefunden wurde, erreichte man jedoch erst durch diese letzte Phase das Optimum selbst.

Auch Garfinkel und Nemhauser/Gar72a/ schlagen ein Verfahren vor, das zur Lösung von Integer-Programmierungsproblemen gedacht ist. Anders als im eben geschilderten Fall soll bei ihnen jedoch auch schon die erste Ausgangs-

lösung vom Algorithmus ermittelt werden. Dazu schlagen sie zwei Vorgehensweisen vor, wie sie schon im letzten Abschnitt angerissen wurden.

Bei der ersten Methode gehen sie von einer trivialen Lösung aus und versuchen schrittweise zu einer zulässigen Lösung zu kommen, indem zu der momentan aktuellen Bezugslösung eine Nachbarschaft ermittelt und anhand eines definierten Maßes die Lösung daraus ausgewählt wird, bei der die Unzulässigkeit minimal ist.

Ein zweiter von ihnen vorgeschlagener Weg, zu einer ersten Lösung zu gelangen, geht von einer mittels eines LP-Verfahrens gefundenen optimalen, aber nicht ganzzahligen Lösung aus. Dazu werden von ihnen Gleichungen generiert, die zusammen einen kleinen Ausschnitt aus dem Lösungsraum um die optimale Lösung herum definieren. Innerhalb dieses Ausschnitts läßt sich nun eine ganzzahlige Lösung finden, die aufgrund der Wahl des Ausschnitts nahe dem nicht ganzzahligen Optimum liegt und alle Restriktionen erfüllt.

Hat man nun, von welchem Ansatz man auch immer ausging, eine erste Lösung gefunden, wird diese dann weiter verbessert. Dafür wird eine Nachbarschaft des Lösungsvektors ermittelt, deren Lösungen sich in einer vorgegebenen Anzahl von Variablen unterscheiden. Für jede dieser Nachbarschaftskandidaten wird mittels eines IP-Verfahrens der geeignete Änderungswert der Variablen ermittelt. Im nächsten Schritt wird aus den so gewonnenen Lösungen diejenige als neue Bezugslösung ausgesucht, die am meisten zur Erreichung des gewünschten Zieles beiträgt. Diese Lösung ist das lokale Optimum bezüglich der gewählten Nachbarschaftsstruktur.

Eine bessere Lösung ist potentiell durch Wahl einer anderen Nachbarschaftsstruktur, d. h. durch Einbeziehen einer anderen Anzahl von Variablen, denkbar. Deshalb wird in einem weiteren Schritt die durch eine solche Iteration zu erwartende Verbesserung gegenüber der schon gefundenen Lösung gegen den durch eine weitere Iteration entstehenden

Rechenaufwand abgewägt und dementsprechend ein neuer Zyklus gestartet oder das Verfahren beendet.

Da auch Garfinkel und Nemhauser die Gefahr sehen, mit ihrem Algorithmus in eine Sackgasse zu laufen, schlagen sie als Möglichkeit zum Ausbruch daraus vor, bestimmte Variablen des momentanen lokalen Optimums zu entfernen, d. h. ihren Wert auf Null zu reduzieren, und mit dieser neuen Ausgangslösung den schon geschilderten Zyklus neu zu beginnen.

Ebenfalls von einer nicht ganzzahligen Optimallösung ausgehend, will Hillier/Hil69a/ eine ganzzahlige Lösung für ein LP-Problem suchen. Da die optimale Lösung gegebenenfalls ein Extremum darstellen kann, in dessen Umgebung man keine gültige ganzzahlige Lösung finden wird, wird in der ersten Phase der von ihm vorgeschlagenen Methode in der Umgebung der durch ein LP-Verfahren gefundenen Lösung nach einer weiteren nicht ganzzahligen Lösung gesucht, die möglichst weiter innen im Lösungsraum liegt und die sich gut auf eine ganzzahlige Lösung runden läßt, ohne dabei die Nebenbedingungen zu verletzen. In der Nähe dieser Lösung wird nun nach einer zulässigen ganzzahligen geforscht.

Zur Verbesserung dieser ersten Integer-Lösung wird im nächsten Schritt des Algorithmusses so lange jeweils eine Variable inkrementiert oder dekrementiert, wie die dabei entstehende Lösung einerseits zulässig bleibt und andererseits besser ist, als die vorhergehende. Daran schließt sich eine Phase an, die dem eben geschilderten Schritt gleich ist, außer daß diesmal verschiedene Änderungen an jeweils zwei Variablen gleichzeitig vorgenommen werden. Diese beiden Schritte wechseln sich nun so lange ab, bis durch keinen von beiden mehr eine Verbesserung erzielt werden kann.

Als Entscheidungskriterium dafür, welche Variable als nächste verändert werden soll, schlägt Hillier zwei Alternativen vor. Im ersten Fall soll sich die Wahl daran orientieren, den Suchprozeß von durch die Restriktionen gegebenen

Grenzen fernzuhalten. Die andere Möglichkeit, die auch schon in den anderen beschriebenen Verfahren angewendet wurde, ist die, die Variable herauszusuchen, die den meisten Gewinn für eine Verbesserung der Zielfunktion bringen wird.

Alle bisher betrachteten Verfahren versuchen, den Zielfunktionswert möglichst schnell in der gewünschten Richtung zu verändern, indem aus den aufgrund des vorgesehenen Algorithmusses ermittelten Kandidaten jeweils der ausgesucht wird, der den besten bis dahin ermittelten Zielfunktionswert ergibt. Müller-Merbach/Mül69a/ hingegen schlägt ein anderes Auswahlverfahren vor, das er das "Verfahren der vorsichtigen Annäherung" nennt. Dabei wird - wie bisher - in jedem Schritt für die maximale ganzzahlige Verbesserung jeder Variablen der Zuwachs des Zielfunktionswertes berechnet. Im Gegensatz zu den bisherigen Verfahren wird jetzt aber die Variable, die die größte Änderung bringt nicht um diesen Maximalwert erhöht oder erniedrigt, sondern jeweils nur um den Wert 1.

Als Begründung für diese Vorgehensweise führt Müller-Merbach aus, daß die Variable mit dem "Greatest Change" besonders stark zur gesuchten Optimallösung beitragen zu können scheint und es daher zweckmäßig ist, ihren Wert zu verändern. Da sich aber der "Greatest Change" in Abhängigkeit vom Lösungspunkt von Variable zu Variable verschiebt, scheint ihm Vorsicht bei der Veränderung der gewählten Variable geboten, um nicht frühzeitig in ein lokales Optimum gedrängt zu werden. Diese Vorsicht mag gerechtfertigt erscheinen, wenn man bedenkt, daß viele der geschilderten Verfahren für den Fall, daß der Suchprozeß in eine Sackgasse, ein lokales Optimum gerät, extra Vorkehrungen vorsehen, um das Verfahren neu aufzusetzen.

Diese vorsichtige Veränderung der Variablenwerte um jeweils eine Einheit hat aber bei Problemen mit großen Variablenwerten eine unnötig hohe Rechenzeit zur Folge. Deshalb wird vorgeschlagen, den Wert der ausgewählten Va-

riablen doch etwas stärker zu verändern, beispielsweise um die gerundete halbe Maximaländerung.

Die gleichzeitige Manipulation mehrerer Variablen kann bei Müller-Merbach nur durch Veränderung neuer, zusätzlich eingeführter Variablen erfolgen, die durch eine Linearkombination der Variablen gewonnen werden, die durch sie repräsentiert werden sollen.

4.2.2. Tauschverfahren

Bokhari/Bok81a/ stellt ein Tauschverfahren vor, daß er für Mapping-Probleme gedacht hat. Er geht davon aus, daß die Anfangslösung dem Algorithmus vorgegeben wird. Im zweiten Schritt wird dann die Nachbarschaft dieser Lösung ermittelt, indem alle paarweisen Tauschmöglichkeiten generiert werden. Aus diesen Nachbarschaftslösungen wird nun diejenige ausgewählt, die den meisten Gewinn bezüglich der Zielfunktion bringt. Der Zyklus von Generieren der paarweisen Vertauschungen und Auswahl der besten der so entstandenen Lösungen wird so lange fortgesetzt, bis der Zielfunktionswert sich nicht mehr verbessern läßt. Damit hat der Algorithmus ein lokales Optimum gefunden. Um dieses eventuell noch verbessern zu können, bietet ein weiterer Schritt des Verfahrens die Möglichkeit, bei Vorliegen bestimmter Bedingungen einige willkürliche Tauschoperationen durchzuführen und die so entstandene Lösung als Ausgangspunkt eines weiteren Zyklusses von Vertauschungen und Verbesserungen gemäß Schritt zwei und drei zu nehmen. Auch das willkürliche Tauschen mit anschließendem erneuten Aufsetzen des Algorithmusses kann beliebig oft durchgeführt werden, solange die dafür notwendigen Bedingungen erfüllt sind.

Für die Lösung von Integer-Programmierungsproblemen ist das von Kreuzberger/Kre69a, Kre70a/ beschriebene Tausch-

verfahren gedacht. Zum Finden einer Ausgangslösung schlägt er vor, ein aus dem ursprünglichen Problem durch Vereinfachung und Hinzufügen von Hilfsvariablen gewonnenes IP-Problem zu lösen. Daran anschließen soll sich, wie schon bei den anderen erwähnten Verfahren, ein Zyklus, in dem zuerst nach Verbesserungen durch Tausch zweier Variablen gesucht wird, von Kreuzberger "einfacher Austausch" genannt, und dann ein "einfacher kombinativer Austausch", in den drei Variablen einbezogen werden, wenn durch keinen einfachen Austausch mehr eine Änderung des Zielfunktionswertes möglich ist. Potentiell stünden einem danach noch diverse "mehrfach kombinatorische Austausche" zur Verfügung, in denen noch mehr Variablen gleichzeitig betrachtet werden. Jedoch sind diese in den meisten Fällen auf eine Kombination einfacher und einfach kombinativer Austausche zurückführbar, so daß Kreuzberger diese komplexeren und daher zunehmend rechenintensiveren Tauschmöglichkeiten nicht für sinnvoll hält.

In jedem Schritt wird wieder aus der Menge der Nachbarschaftslösungen diejenige ausgewählt, die den Zielfunktionswert am meisten in die gewünschte Richtung verändert. Nach jeder Änderung beginnt sofort ein neuer Zyklus, d. h. es wird wieder erneut überprüft, ob nun ein einfacher Austausch das bisherige Ergebnis verbessern kann.

Reiter und Sherman/Rei65a/ suchen sich eine beliebige, zufällige Lösung zum Start ihres Verfahrens aus. Darauf aufbauend werden verschiedene weitere Vorgehensweisen geschildert. Im einfachsten Fall wird ein paarweiser Tausch benachbarter Variablen betrachtet und die Bezugslösung immer sofort durch den Nachfolger-Kandidaten ersetzt, wenn dieser einen besseren Zielfunktionswert hat. Varianten dazu sind, zuerst eine bestimmte Anzahl an Kandidaten zu ermitteln - vorgeschlagen werden sechs oder alle - und aus diesen als Nachfolger den bezüglich des Zielfunktionswerts besten auszuwählen. Ebenfalls wird hier die Möglichkeit aufgezeigt, eine Nachbarschaft zu durchsuchen, die sich durch einen Tausch von mehr als zwei Variablen ergibt.

4.2.3. Zusammenfassung

Nach der Betrachtung dieser hier beispielhaft herausgegriffenen Algorithmen läßt sich zusammenfassend sagen, daß die Tauschverfahren und die Verfahren des direkten Suchens, derer sich diese Algorithmen bedienen, nicht garantieren können, daß sie fähig sind, zu jeder beliebigen aktuellen Aufgabenstellung des Problemkreises, für den sie konzipiert wurden, eine approximative Lösung zu finden. Die Verfahren sind so gestaltet, daß sie sich als sehr leistungsfähig erweisen, wenn es gilt, eine gute zulässige Lösung zu finden und diese dann zu verbessern. Deshalb kann manchmal der Versuch, eine Anfangslösung mit diesen Verfahren zu finden, fehlschlagen, da mit ihnen nicht notwendigerweise jede zulässige Lösung ermittelt werden kann. Dies trifft insbesondere auf Aufgabenstellungen zu, die durch die Nebenbedingungen sehr restriktiv eingeschränkt werden.

Alle Autoren berichten von erstaunlich guten Ergebnissen, die sich bei diversen Tests im Normalfall ergaben, sowohl was die Qualität der Lösung, als auch was den Rechenzeitbedarf betrifft, der zu ihrer Ermittlung benötigt wurde. Die geschilderten Erfahrungen ermutigen zu der Annahme, daß diese Verfahren auch für sehr große Probleme, die, vom benötigten Rechenaufwand her gesehen, weit über den Möglichkeiten optimaler Algorithmen liegen, mit vernünftigem Rechenzeitbedarf einsetzbar sind. Mehr noch, die ermittelten Lösungen wurden nicht nur sehr schnell gefunden, sondern waren zudem auch noch sehr gut. In den meisten Fällen wurde sogar das Optimum ermittelt. Aber selbst wenn die Lösung nicht optimal war, wich sie vom Optimum meist nur sehr geringfügig ab.

5. Vergleichstests von Tauschverfahren bei Scheduling-Problemen

5.1. Motivation

Nach der allgemeinen Betrachtung über mögliche Algorithmen sei nun wieder zu dem hier interessierende Scheduling-Problem zurückgekehrt.

Das Zuordnen von mehreren Programmen zu mehreren Rechnern ist, wie in Kapitel 2 geschildert, NP-vollständig. Daher werden exakte, optimierende Verfahren nur in Ausnahmefällen sinnvoll einsetzbar sein. In der Tat hat die Betrachtung der verschiedenen optimalen Verfahren gezeigt, daß diese für die hier interessierende Problematik nicht brauchbar sind.

Viele der in der Literatur vorgeschlagenen Algorithmen gehen von einer sehr idealisierten Vorstellung der Problemumgebung aus, z. B. daß die Prozessoren unbeschränkt große Speicherkapazität haben. Vielfach lassen sich in die Verfahren keine anderen Kriterien zur Optimierung mit einbeziehen als der Kommunikationsaufwand, der von Programmen herrührt, die auf verschiedenen Rechnern laufen, und allenfalls noch die von den einzelnen Programmen benötigte Rechenzeit. Aber selbst bei diesen Verfahren wird üblicherweise von einer homogenen Hardware ausgegangen. Heutzutage ist es jedoch immer mehr gang und gäbe, beliebige, nicht notwendigerweise gleiche Rechnerkonfigurationen zusammenzuschalten. Andererseits beschränken die Verfahren die Anwendungsmöglichkeiten oft dadurch, daß sie nur die Zuordnung zwischen zwei Prozessoren, allenfalls noch dreien, zulassen.

Bei den Methoden, in die alle hier interessierenden Eigenschaften und Restriktionen eines Systems mit einfließen können, erweist sich der zur Lösung einer speziellen Problemstellung benötigte Rechenzeitaufwand als unabschätzbar. Man muß also immer mit dem schlechtesten Verhalten rechnen. Die "worst-case"-Schränken der Algorithmen

sind diesbezüglich aber nicht ermutigend. Sieht man sich deshalb das durchschnittlicherweise zu erwartende Verhalten dieser Verfahren an, welches ja im Normalfall das wichtigere ist, so muß man sehen, daß selbst da mit enormem Aufwand gerechnet werden muß. Wie schon erwähnt, liegt bei den meisten Algorithmen der hauptsächlichste Aufwand nicht darin, überhaupt eine optimale Lösung zu finden, sondern die Optimalität der gefundenen Lösung nachzuweisen.

Es scheint deshalb sinnvoll zu sein, sich eine Unterklasse von Scheduling-Problemen zu definieren und für diese Probleme ein geeignetes heuristisches Lösungsverfahren zu suchen, daß in angemessener Zeit eine Lösung ermittelt, die möglichst optimal ist. Diese Unterklasse sollte nicht zu umfassend sein, damit genügend problemspezifische Details in den Entwurf der Heuristik eingebracht werden können, um so zu einem effizienten Verfahren zu gelangen. Andererseits sollte der mit dem resultierenden Algorithmus bearbeitbare Problemkreis auch nicht zu restriktiv eingeschränkt werden, damit das Verfahren nicht nur für einige wenige Aufgabenstellungen einsetzbar und somit praktisch uninteressant ist.

Herausgegriffen wurde in dieser Arbeit die Gruppe von Problemen, bei denen es um die Zuordnung eines Aufgabensystems auf ein Multiprozessorsystem geht. Die Hardware des Systems kann dabei beliebig, homogen oder heterogen, sein. Das Aufgabensystem soll statisch und deterministisch sein. Außerdem ist gedacht, das Verfahren besonders bei solchen Aufgabensystemen einzusetzen, die periodisch immer wieder neu abgearbeitet werden und bei denen es sehr darauf ankommt, so optimal wie möglich die einzelnen Programme auf die Rechner zu verteilen. Solche Systeme lassen sich in der Praxis häufig in der Prozeßsteuerung und -überwachung finden, aber auch bei anderen Anwendungen wie beispielsweise der Simulation. Hier lohnt es sich, doch etwas Aufwand zu treiben, um eine Lösung zu finden, die möglichst eine optimale ist, aber wenn schon nicht das, so doch wenigstens nicht weit vom Optimum entfernt.

Die im weiteren geschilderte Vorgehensweise ist nicht auf diese Art von Programmsystemen beschränkt und kann prinzipiell auch für die Zuordnung einmal durchlaufener Programmpakete verwendet werden. Jedoch wird sich bei solcher Problemstellung oft der Aufwand zum Erreichen einer Lösung in unmittelbarer Nähe des Optimums nicht lohnen. Man wird meistens schnellere Verfahren wählen, selbst wenn die von ihnen gefundenen Lösungen etwas weiter vom Optimum entfernt sind, als bei den im weiteren geschilderten Verfahren. Dies ist auch deshalb hier normalerweise sinnvoller, weil in Systemen mit einmal durchlaufenen Aufgabenpaketen oft neue Programmpakete verteilt werden müssen und daher ein schnellerer Algorithmus einem optimaleren vorzuziehen ist.

In Kapitel 3.2. wurde gezeigt, daß es nicht immer möglich sein wird, ein exaktes Verfahren mit heuristischem Abbruchkriterium zur Approximation der optimalen Lösung anzugeben. Die Kenntnis des wirklichen Optimums wird üblicherweise nicht vorliegen. Daher ist es oft schwierig, einen Wert als absolute Abweichung vom gewünschten optimalen Zielfunktionswert anzugeben. In manchen Fällen wird es jedoch gelingen, das Optimum in irgendeiner Form näherungsweise zu berechnen.

Ist es beispielsweise das Ziel, die Programme so zu verteilen, daß alle Prozessoren möglichst gleichmäßig ausgelastet sind, so läßt sich der optimale Rechenbelastungswert der einzelnen Rechner dadurch annähern, daß man eine mittlere Belastung ausrechnet, also die Rechenzeiten aller Programme aufsummiert und durch die Anzahl der zur Verfügung stehenden Prozessoren teilt. Dadurch hat man eine ungefähre Vorstellung, wie der optimale Wert aussehen wird, muß sich aber dessen bewußt sein, daß der so errechnete Mittelwert im Einzelfall sehr stark vom wirklichen Optimum abweichen kann, z. B. wenn ein sehr rechenintensiver Prozeß in dem Aufgabenpaket ist, während die anderen im Verhältnis zu ihm nur sehr wenig Rechenzeit benötigen. In solchen Fällen wird die Approximation erheblich verfälscht. Dies hat zur Folge, daß

eventuell eine suboptimale Lösung innerhalb eines gewissen Bereichs um den errechneten Näherungswert herum gesucht wird, obwohl selbst die optimale aufgrund des rechenintensiven Prozesses niemals in diesem Bereich liegen kann.

Beim Entwurf eines Algorithmusses, der sich solcher Schranken zum Abbruch eines Verfahrens bedient, muß deshalb darauf geachtet werden, daß entsprechende Vorkehrungen getroffen werden, damit nicht unnötig langer Rechenaufwand entsteht oder das Verfahren gar ohne Lösung abbricht.

Läßt sich nun solch eine Abschätzung des Optimalwertes angeben, so können z. B. vielfach "Branch and Bound"-Verfahren trotzdem nicht eingesetzt werden, weil sich keine Kriterien angeben lassen, um vernünftige Schranken für die Teillösungen zu berechnen.

Die Approximation des Optimalwertes durch Verwendung exakter Verfahren mit heuristischen Abbruchkriterien scheint aus diesen Gründen nicht sinnvoll. Die Einsatzmöglichkeit bei dem ins Auge gefaßten Anwendungsgebiet wäre zu eingeschränkt. Je nachdem, welcher Struktur das aktuelle Problem, die augenblicklich relevanten Restriktionen und das gewünschte Zielkriterium gerade ist, müßten andere Verfahren oder andere Abbruchkriterien zur Lösung eingesetzt werden. Dies würde darauf hinauslaufen, daß dem Benutzer eine Verfahrensbibliothek zur Verfügung gestellt werden müßte. Für den Anwender, der sich jedoch nicht genau mit den Feinheiten jeder der darin enthaltenen Methoden auskennt, wird es aber schwer sein, die jeweils gerade beste daraus auszuwählen.

Deshalb erscheint es am besten, eine heuristische Vorgehensweise zu verwenden, die allgemein genug ist, um alle Aufgabenstellungen des gewünschten Problemkreises behandeln zu können, die aber trotzdem die Approximationslösung in vertretbarer Zeit liefert. Solch eine allgemeine Vorgehensweise hat zudem noch den Vorteil, daß man solch ein Verfahren automatisieren kann, so daß der Anwender nur noch die

aktuellen Parameter seines Problems eingeben muß, sich aber in keiner Weise um irgendwelche Besonderheiten des Verfahrens zu kümmern braucht.

Die Tauschverfahren sind eine solche Vorgehensweise, die es gestatten, beliebige Kriterien, Restriktionen und Besonderheiten eines vorliegenden Systems einzubeziehen. Im vorhergehenden Kapitel wurde schon erwähnt, daß dies recht leistungsfähige Algorithmen sind, die in bemerkenswert schneller Zeit Lösungen finden, die nur unwesentlich von der optimalen abweichen. Erstaunlicherweise fanden sich in der durchsuchten Literatur keinerlei Hinweise auf einen Einsatz dieser Methoden bei Scheduling-Problemen. Ebenfalls fanden sich keine Angaben über experimentelle und vergleichende Ergebnisse zwischen den verschiedenen Verfahrensvarianten. Dies regte zu einer genaueren Untersuchung dieser Verfahren an mit dem Ziel, ihre Einsatzfähigkeit für den betrachteten Problemkreis zu prüfen.

5.2. Vergleichstests

5.2.1. Voraussetzungen

Für ein effizientes Tauschverfahren spielt das Kriterium zur Auswahl der nächsten Bezugslösung aus der Nachbarschaft der momentanen eine entscheidende Rolle. Bei den in der Literatur vorgestellten Verfahren wird überwiegend jene Lösung als weiterer Ausgangspunkt gewählt, die den Zielfunktionswert von allen Lösungen der Nachbarschaft am meisten in die gewünschte Richtung vorantreibt. Dagegen wird diesbezüglich auch von anderer Seite warnend bemerkt, daß dabei die Gefahr besteht, zu schnell in eine Sackgasse zu geraten, d. h. ein lokales Optimum zu finden, in dessen Nachbarschaft sich keine bessere Lösung befindet, die aber durch vorsichtigeres Taktieren hätte umgangen werden können zu gunsten eines anderen, besseren, lokalen Optimums oder

gar der optimalen Lösung selbst. Müller-Merbach/Mül69a/, der ein Vertreter dieser Ansicht ist, hält es deshalb für sinnvoll, den Zielfunktionswert immer nur in kleinen Schritten zu verändern, um es zu vermeiden, frühzeitig in solche Sackgassen zu laufen.

Um zu sehen, wie die Tauschverfahren bei dem Problemkreis des Scheduling auf die verschiedenen Auswahlkriterien reagieren, wurde ein Dialogsystem implementiert, in dem die verschiedensten Methoden zum Finden einer ersten Lösung und die diversen Varianten der Tauschverfahren miteinander kombiniert werden können. Es besteht aus einem Modul als Schnittstelle zum Anwender, der die Kommandos entgegen nimmt, interpretiert und einen die gewünschte Funktion ausführenden weiteren Modul als Unterprozeß startet. Es läuft zur Zeit auf einer Perkin Elmer 3210 unter dem Betriebssystem UNIX* und ist in der Programmiersprache C implementiert. Dadurch ist eine Portierung auf andere Rechner, für die ein C-Compiler zur Verfügung steht, unproblematisch.

Der Anwender hat die Möglichkeit, die Daten beliebiger Aufgaben- und Prozessorsysteme im Dialog einzugeben. Auf Wunsch werden diese in Dateien hinterlegt, von denen sie für spätere Testläufe wieder abrufbar sind.

Eine erste Lösung kann entweder im Dialog vorgegeben werden, oder man läßt sie von einer der bereitgestellten Methoden (z. B. LPTF) ermitteln. Die gefundene Lösung wird dem Benutzer protokolliert.

Er hat nun die Möglichkeit, selbst bestimmte Aufgaben auf andere Rechner zu verlagern oder Tauschpartner zu bestimmen. Der entsprechende Modul überprüft jeweils die Restriktionen, führt die gewünschte Operation aus, falls keine Nebenbedingung verletzt ist, und gibt die neue Lösung

*) UNIX ist ein eingetragenes Warenzeichen von Bell Lab.

als Quittung aus. Konnte wegen Verletzung einer Restriktion keine Änderung der Zuordnung vorgenommen werden, wird eine diesbezügliche Fehlermeldung ausgegeben.

Will der Benutzer eine Lösung nicht explizit verändern, kann er eine der vielen angebotenen Tauschverfahren aufrufen. Wiederum wird ihm die neue Endlösung mitgeteilt, worauf er weitere Verfahren starten oder selbst Änderungen vorgeben kann.

Zu jedem Zeitpunkt besteht die Möglichkeit, die Bearbeitung eines Problems abubrechen und den bis zu diesem Zeitpunkt erreichten Stand bei der Lösungsfindung abzuspeichern, um später dort wieder neu aufsetzen zu können.

Zur Abschätzung der Qualität der ermittelten Endlösung kann auch ein optimales Verfahren zum Vergleich gestartet werden, wobei man aber aufgrund der NP-Vollständigkeit mit längeren Wartezeiten am Terminal rechnen muß.

Als erstes wurde einige Zeit mittels dieses Dialogsystems experimentiert, um geeignete Kombinationen der Tauschverfahren zu finden. Auf dieser Grundlage wurde dann ein Verfahren implementiert, daß von einer vorgegebenen Lösung ausgeht, zuerst paarweise Tausche vornimmt, bis sich damit keine Verbesserung mehr erzielen läßt, danach nach Tauschpartnern im Sinne von "zwei gegen einen" sucht, d. h. zwei Prozesse, die beide demselben Rechner zugewiesen werden sollten, werden gegen einen einem anderen Prozessor zugeordneten Prozeß getauscht. Konnten solche Tauschoperationen vollzogen werden, beginnt der Zyklus von neuem, d. h. es wird wieder nach paarweisen Tauschmöglichkeiten gesucht, anschließend von neuem nach "zwei gegen einen"-Tauschen. Ist ein Tausch auf keine dieser Arten mehr möglich, wird noch eine weitere Phase angeschlossen, in der nun für die zwei Prozesse des einen Rechners ebenfalls ein Paar von Prozessen eines anderen Rechners gesucht wird, um bei erfolgreicher Suche von neuem den gesamten Zyklus zu beginnen.

Dieser allgemeinen Vorgehensweise können nun verschiedene Entscheidungskriterien vorgegeben werden, um einerseits eine bestimmte Nachbarschaft zu der aktuellen Bezugslösung auszuwählen, und andererseits um aus der so bestimmten Nachbarschaft die nächste Lösungen zu ermitteln.

Hat man eine Bezugslösung ermittelt, so hat man verschiedene Möglichkeiten, die Nachbarschaft dazu festzulegen. Diese werden durch die Reihenfolge bestimmt, in welcher die anderen Prozessoren auf der Suche nach Tauschpartnern berücksichtigt werden sollen. Dabei kann man folgenderweise vorgehen:

- A 1. Man nimmt immer den erst besten Prozessor, was bedeutet, daß zum Implementierungszeitpunkt irgendeine Reihenfolge vorgegeben wird.
- A 2. Man betrachtet immer denjenigen Prozessor als nächstes, der bisher noch am wenigsten belastet ist bezüglich des Optimierungskriteriums.
- A 3. Man rechnet jeweils eine aktuelle mittlere Belastung aus und sucht bei demjenigen Rechner als nächstes nach Tauschpartnern, der am besten dieser mittleren Belastung genügt.
- A 4. Ebenfalls steht einem auch die Umkehrung der zweiten Variante offen, also den Prozessor als nächstes zu berücksichtigen, der die augenblicklich nächstgrößte Rechenlast zu tragen hat. Ob dies sinnvoll ist, werden die nachfolgenden Testergebnisse zeigen.

Wenn man auf eine der genannten Arten eine Nachbarschaft festgelegt hat, muß beim Entwurf des Algorithmusses als nächstes entschieden werden, wie aus der Menge der dadurch bestimmten potentiellen Nachfolgerlösungen eine als neuer Bezugspunkt ausgewählt wird. Das Entscheidungskriterium wird sich in diesem Fall an dem Zielfunktionswert

orientieren. Die sich für die Wahl anbietenden Alternativen sind ähnlich denen für die Auswahl der Nachbarschaft selbst:

- B 1. Die einfachste und schnellste Möglichkeit ist es, diejenige Lösung als Nachfolger zu wählen, die als erste einen Zielfunktionswert hat, der größer ist als der der augenblicklich aktuellen Bezugslösung.
- B 2. Andererseits kann man auch zuerst alle Nachbarschaftslösungen explizit überprüfen und daraus diejenige auswählen, die die größte Änderung des Zielfunktionswertes bewirkt, in der Hoffnung, auf diese Weise am schnellsten zum Ziel zu kommen.
- B 3. Wie schon erwähnt, kann eine Veränderung des Zielfunktionswertes um den größtmöglichen Wert eventuell in Sackgassen führen, die bei vorsichtigerem Taktieren zu gunsten einer besseren (suboptimalen) Endlösung hätten umgangen werden können. Daher kann man auch statt der Nachbarschaftslösung mit dieser maximalen Änderung jene als Nachfolger aussuchen, die eine minimale Veränderung des Zielfunktionswertes bewirkt.
- B 4. Befürchtet man, mit der Variante 2 zu schnell in ein schlechtes Suboptimum abgedrängt zu werden, und ist einem das Vorgehen der dritten Alternative zu vorsichtig, kann man auch unter allen Nachbarschaftslösungen eine mittlere Änderung des Zielfunktionswertes berechnen und diejenige Lösungen auswählen, die diesem Mittelwert am nächsten kommt.

Kombiniert man nun die vier Alternativen zur Wahl der Nachbarschaft mit den vier Varianten, aus einer solchen Nachbarschaft die Nachfolgerlösung zu bestimmen, erhält man sechzehn Möglichkeiten für die Vorgehensweise eines Tauschalgorithmusses. Etliche Testbeispiele wurden von diesen sechzehn Varianten bearbeitet und die von ihnen ermittelten Endlösungen gegeneinander verglichen.

Als Problemstellung wurde den Beispielen die schon mehrfach skizziert Aufgabenstellung des Zuordnens von n Programmen zu m Prozessoren zugrundegelegt.

Als Hardware wurde von einem homogenen Multiprozessor-system ausgegangen, bei dem jeder Prozessor mehrere verschiedene Speicher besitzt, deren Größe sehr begrenzt ist. Die so festgelegte Aufgabenstellung erfolgte in Anlehnung an die im Nachfolgenden noch näher beschriebenen Anforderungen bei dem Projekt "MUPSI" (siehe Kapitel 6.), ist jedoch eigentlich willkürlich und ohne Beschränkung der Allgemeinheit. Dasselbe gilt für die den verschiedenen Beispielen zugrunde liegenden aktuellen Daten für die geschilderte Problematik.

Die Programme bearbeiten ihnen gelieferte Daten und geben diese eventuell verändert an nachfolgende Programme weiter. Das gesamte Aufgabensystem soll periodisch ablaufen, so daß die Daten pipeline-artig durch das System geschleust werden. Dies bedeutet, daß die Programme im Prinzip voneinander unabhängig sind. Da jedoch ein Pipeline-Betrieb vorgesehen ist, muß jedes Programm allezeit seine sämtlichen Anforderungen z. B. an Speicherplatz befriedigt bekommen.

Die Länge einer Bearbeitungsperiode wird vom dem Prozessor bestimmt, der die meiste Zeit zur Bearbeitung der ihm zugewiesenen Programme braucht. Daher war es das Optimierungsziel, die Prozessoren möglichst gleichmäßig auszulasten, d. h. dieses Maximum der Summen von Rechenzeitanforderungen je Prozessor zu minimieren.

Es wurde versucht, durch die Beispieldaten die verschiedensten Situationen abzudecken. So wurden einerseits Testfälle mit wenigen und solche mit vielen einzelnen Programmen durchgerechnet, die die verschiedensten Anforderungen an die unterschiedlichen Speichertypen stellten, wobei jedoch jede Anforderung für sich weit unter der maximal möglichen lag. Andererseits wurden jedoch auch solche

Fälle betrachtet, in denen der Bedarf eines Programms (oder auch von mehreren) bezüglich eines oder mehrerer Speicher groß war (bis hin zur Anforderung des gesamten Speichers zumindest einer Art) so daß auf jeden Fall nicht mehr jedes andere Programm mit auf demselben Rechner laufen konnte.

Jedem Testbeispiel lag eine vorgegebene Ausgangssituation zugrunde. Jedes Beispiel wurde mit allen sechzehn Strategie-Varianten durchgespielt (selbstverständlich jeweils mit derselben Ausgangslösung). Wo es zeitlich irgend möglich war, wurde auch das wirkliche Optimum zu Vergleichszwecken mittels eines "backtracking"-Algorithmusses ermittelt.

Die genauen Daten einiger der Testbeispiele, die den nachfolgend dargestellten Ergebnissen zugrunde liegen, sind als Anschauung im Anhang aufgeführt.

5.2.2. Ergebnisse

Durch die Tests bestätigte sich die Vermutung, daß die Tauschverfahren für den Einsatz bei dem geschilderten Problemkreis sehr gut geeignet sind. Ja noch mehr, die erzielten Resultate übertrafen dabei die Erwartungen bei weitem.

Beim Einsatz approximativer Verfahren gilt es normalerweise schon als gut, wenn man die optimale Lösung mit einer maximalen Abweichung von bis zu 10 % annähern kann. Sehr gut kann man einen Algorithmus nennen, wenn die Abweichung des Suboptimums vom Optimum 5 % nicht übersteigt.

Bei den Tauschverfahren zeigte es sich jedoch, daß alle sechzehn Varianten Lösungen ermittelten, die nahezu immer ein Optimum selbst waren. Falls ein Suboptimum gefunden wurde, betrug die Abweichung von der optimalen Lösung weniger als 0,5 %. Bei allen durchgerechneten Beispielen gab es nur einen Fall, bei dem zwei der sechzehn Strategien

als bestes eine Lösung fanden, die eine größere Abweichung hatte (12 %).

Da sich die Qualität der Lösungen bezüglich des gewählten Optimierungskriteriums bei allen Varianten und allen Beispielen nahezu gleich zeigte, bleibt nun noch zu untersuchen, wie sich die verschiedenen Strategien bezüglich des Aufwands zur Ermittlung einer Lösung für eine aktuelle Problemstellung verhalten.

In der Natur der Tauschverfahren liegt es, daß sie keinerlei Zwischenergebnisse aufheben müssen. Jede neue Bezugslösung ergibt sich unmittelbar aus der vorhergehenden. Rückwärtsschritte (wie beispielsweise bei den "Branch and Bound"-Verfahren) sind nicht vorgesehen. Somit ist der Speicherbedarf der Verfahren direkt proportional der Problemgröße, und zwar unabhängig von den verwendeten Entscheidungskriterien für die Ermittlung einer Nachbarschaft und für die Auswahl einer Lösung daraus. Bei der Zuordnungsproblematik heißt dies, daß der Speicherbedarf linear von der Anzahl n der zu verteilenden Programme und der Zahl m der vorhandenen Prozessoren abhängt, also von der Ordnung $O(n \cdot m)$ ist.

Sei nun noch den Aufwand an Rechenzeit betrachtet, den die einzelnen Strategien zur Ermittlung der von ihnen gelieferten Lösung benötigen.

Es stellte sich heraus, daß der Wahl des Entscheidungskriteriums, das aus den Nachbarschaftslösungen den Nachfolger extrahiert, eine entscheidende Rolle zukommt. Ganz gleich, in welcher der genannten Reihenfolgen die Prozessoren zur Bestimmung der Nachbarschaft berücksichtigt wurden, jedes Mal erwies sich die Strategie als die beste, bei der einfach die erste Lösung mit einem besseren Zielfunktionswert als der der augenblicklich aktuellen Bezugslösung ausgewählt wurde (Variante B 1). Das zweitbeste Kriterium war jenes, das auch in den meisten in der Literatur

zitierten direkten Such- und Tauschverfahren verwendet wird, nämlich als nächste Bezugslösung diejenige heranzuziehen, die von allen Nachbarschaftslösungen die größte Änderung des Zielfunktionswertes bewirkt (Variante B 2). Als drittbestes Auswahlmerkmal stellte sich das heraus, bei dem die mittlere Änderung aller Nachbarschaftslösungen berechnet wird und dann diejenige Lösung als Nachfolger der momentan aktuellen bestimmt wird, die diesem Mittelwert am nächsten kommt (Variante B 4). Am schlechtesten schnitt bei allen Vergleichen die vorsichtigste Vorgehensweise ab (Variante B 3). Durch die Auswahl der Nachbarschaftslösung, die den Zielfunktionswert am wenigsten verändert, wurde am meisten Rechenzeit benötigt.

Diese Reihenfolge der Varianten bezüglich des von ihnen benötigten Rechenzeitaufwands war zu erwarten. Da bei Variante B 1 meistens nur ein geringer Teil aller Nachbarschaftslösungen explizit geprüft werden muß, spart man dabei sehr viel Zeit. Nur im allerletzten Zyklus, wo keine Verbesserung des Zielfunktionswertes mehr möglich ist, müssen auch hier alle Lösungen betrachtet werden. Dies ist bei allen der anderen drei Varianten anders. Bei ihnen müssen immer erst alle Lösungen ermittelt werden, um danach entscheiden zu können, welche die größte, mittlere oder geringste Änderung des Zielfunktionswertes bewirkt. Bezüglich dieses Aufwands verhalten sich daher diese drei Varianten nahezu gleich (bei Variante B 4 kommt noch ein kleiner Aufwand zur Bestimmung des Mittelwertes zusätzlich dazu). Hier wird daher der insgesamt zu erbringende Rechenaufwand dadurch bestimmt, wie schnell der Zielfunktionswert gegen das Optimum konvergiert. Naturgemäß geschieht dies desto schneller, je größer die Änderungen bei jedem einzelnen Iterationsschritt sind. Die sich in den Tests gezeigte Reihenfolge ist ein zwangsläufiges Ergebnis dieser Gesetzmäßigkeit.

Wie groß sind nun aber diese Unterschiede im Rechenaufwand bei den einzelnen Varianten? Sind sie so gravierend, daß man sie überhaupt berücksichtigen muß, oder ist es

doch mehr oder weniger gleich, welche Variante man implementiert?

Die durchgeführten Tests ergaben, daß die beste Lösung, sowohl bezüglich der Nähe zum Optimum, als auch hinsichtlich des zu ihrer Ermittlung nötigen Rechenaufwands, immer von einer Strategie gefunden wurde, die die Auswahl unter den Nachbarschaftslösungen anhand des Entscheidungskriteriums B 1 traf. Der für die anderen Varianten erforderliche Mehraufwand ist aus der Tabelle 1 ersichtlich.

	mindestens	maximal	im Durchschnitt
Kriterium B 1	- 7 %	92 %	17 %
Kriterium B 2	18 %	377 %	113 %
Kriterium B 3	24 %	562 %	170 %
Kriterium B 4	18 %	495 %	135 %

Tabelle 1: Mehraufwand an Rechenzeit bezüglich des Entscheidungskriteriums B (Kriterium A beliebig) gegenüber dem Algorithmus, der am schnellsten die beste Lösung gefunden hat

Für alle in Tabelle 1 enthaltenen Angaben wurden bei allen Varianten die Fälle, bei denen die benötigte Rechenzeit weit über dem Normalen lag, für die Ermittlung des prozentualen Verhältnisses nicht berücksichtigt, um das Bild nicht durch Extremfälle zu verfälschen. Nur bei den Varianten mit dem Kriterium B 1 zeigten sich keinerlei von der Norm abweichende Bedürfnisse an zur Ermittlung der Lösung

notwendigen Rechenzeit. Die Zahl der diesbezüglich völlig aus dem Rahmen fallenden Resultate betrug bei den Varianten, die die Kriterien B 2, B 3 oder B 4 beinhalteten, jeweils ca. 2 % aller Testbeispiele. In diesen Fällen überstieg der Aufwand jegliche vernünftige und zumutbare Grenze. Bei dem krassesten Beispiel war der Algorithmus nach fast acht Stunden CPU-Zeit (auf einem Rechner des Typs 3210 von Perkin Elmer) immer noch nicht fertig, während üblicherweise eine Lösung z. B. von den Algorithmen, die das Kriterium B 1 beinhalteten, in einigen Sekunden gefunden wurde, innerhalb weniger Minuten aber von allen getesteten Verfahrensvarianten.

Etwas erstaunen mag die Angabe von -7 % als Mindestzeit bei Kriterium B 1. Dies ergibt sich daher, daß als Bezugspunkt jeweils der Algorithmus ausgewählt wurde, der von allen Verfahren, die die optimale Lösung lieferten, die kürzeste Rechenzeit benötigte. Bei den Vorgehensweisen, die Entscheidungen anhand des Kriteriums B 1 trafen, gab es aber durchaus Fälle, bei denen diese Algorithmen zwar das Optimum knapp verfehlten, aber dafür schneller fertig waren als das Verfahren (eben um ca. 7 %), welches das wirkliche Optimum ermittelte.

Die Varianten, die B 2, B 3 oder B 4 beinhalteten, benötigen also im Durchschnitt etwa die 7- bis 10-fache Rechenzeit wie die Verfahren mit B 1. Die Unterschiede sind also doch so beträchtlich, daß die Wahl des Kriterium zur Auswahl der neuen Bezugslösung aus den Nachbarschaftslösungen gut zu überlegen ist. Da sich zeigte, daß alle Alternativen bezüglich der Qualität der Lösung, also der Nähe zum Optimum, gleich sind, sollte sich die Wahl des Verfahrens an dem benötigten Rechenaufwand orientieren. Aufgrund der eben dargestellten Ergebnisse ist somit eindeutig die Entscheidungsalternative B 1 zu bevorzugen; man nimmt also am besten immer die erste Lösung, mit einem besseren Zielfunktionswert.

Mit der Festlegung des Kriteriums B wurde die Anzahl der Varianten auf vier beschränkt, die sich nur noch darin unterscheiden, in welcher Reihenfolge die Prozessoren auf der Suche nach Tauschpartnern berücksichtigt werden. Wie wirkt sich nun dieses Kriterium A im einzelnen aus?

Die optimale Lösung wurde in der Hälfte aller Testfälle von dem Algorithmus gefunden, der die Nachbarschaft mittels des Kriteriums A 1 bestimmte, also im Prinzip die Prozessoren in einer willkürlichen, aber festen Reihenfolge betrachtete.

Am zweithäufigsten fand sich die in schnellster Zeit ermittelte optimale Lösung mittels des Verfahrens, das immer denjenigen Prozessor als nächsten berücksichtigte, der die bis dahin geringste Auslastung hatte (Kriterium A 2). Dies erscheint auch sinnvoll, wenn man bedenkt, daß in diesem Fall von dem am stärksten belasteten Prozessor ein oder mehrere Programme gegen andere, aber mit insgesamt weniger Rechenzeitanforderungen, ausgetauscht werden. Die Last des am stärksten ausgelasteten Rechners wird also verringert, während die des am wenigsten belasteten gleichzeitig erhöht wird. Jedoch war unter den mit diesem Algorithmus (Variante A 2 / B 1) gefundenen Resultaten auch die Lösung, die die größte Abweichung vom Optimum hatte (12 %).

Die dritte Variante (mit A 3) fand in allen Fällen die optimale Lösung, brauchte dazu aber immer erheblich länger, als die anderen Verfahren.

Von den vier jetzt betrachteten Verfahren wurde von der Alternative A 4 / B 1 die zur Bestimmung zeitaufwendigste Lösung ermittelt.

Einen genaueren Überblick über den zum Finden der Lösung benötigten Rechenaufwand dieser vier Verfahren gibt Tabelle 2. Dabei wurde bei der Variante mit A 4 der aus dem Rahmen fallende Extremfall hinsichtlich des Rechenzeitbedarfs zwar als Maximum erwähnt, in die Berechnung des Durchschnitts aber nicht mit einbezogen, da alle anderen

Fälle weniger als den halben zusätzlichen Aufwand benötigten.

	mindestens	maximal	im Durchschnitt
Kriterium A 1	- 7 %	9 %	1 %
Kriterium A 2	- 7 %	30 %	5 %
Kriterium A 3	18 %	42 %	31 %
Kriterium A 4	0 %	92 %	16 %

Tabelle 2: Mehraufwand an Rechenzeit bezüglich des Entscheidungskriteriums A (Kriterium B = B 1) gegenüber dem Algorithmus, der am schnellsten die beste Lösung gefunden hat

Von alle Kombinationen erwies sich also diejenige bei allen Testbeispielen als die günstigste, in der die Entscheidungsalternativen A 1 und B 1 vereint war.

Dies läßt sich nur dadurch erklären, daß in diesem Fall auf der Suche nach möglichen Tauschpartnern einmal ein weniger belasteter Prozessor und einmal ein stärker belasteter als nächster überprüft wurde, wodurch die Vorteile des Kriteriums A 2 mit denen des Kriteriums A 4 kombiniert werden. Mit dem Kriterium A 2 wird das Ziel der gleichmäßigen Auslastung mit jedem Iterationsschritt am meisten gefördert. Das gesamte Verfahren konvergiert somit hierbei am schnellsten. Die Variante mit dem Kriterium A 4 ist das vorsichtigste Verfahren, da hier die potentiell kleinsten Schritte hinsichtlich der Konvergenz gemacht werden. Der

Ausgleich zwischen A 2 und A 4 ist im Prinzip auch das Ziel der Mittelwert-Variante A 3. Jedoch wird mit A 1 eine gewisse Ausgewogenheit erzielt, ohne den Aufwand zur expliziten Berechnung eines derartigen Mittelwertes treiben zu müssen.

Bei den anderen Kombinationen mit B 1 traten außerdem (wie schon erwähnt) entweder Extremfälle bezüglich der Qualität der gefundenen Endlösung (bei A 2) oder des Aufwands zu ihrer Ermittlung auf (bei A 4) oder es zeigte sich ein generell mäßiges Zeitverhalten (A 3).

In einer zweiten Testserie wurde die Zielvorstellung dahingehend erweitert, daß als weiteres Optimierungskriterium der durch die jeweilige Lösung hervorgerufene Kommunikationsaufwand berücksichtigt wurde. Das Kriterium A zur Auswahl der Nachbarschaft blieb also unverändert.

Hingegen wurde bei Variante B 1 nicht diejenige Nachbarschaftslösung als Nachfolger bestimmt, die als erste einen besseren Zielfunktionswert brachte, sondern diejenige die als erste ausgewählt, die entweder hinsichtlich der Auslastung der Prozessoren besser war, oder diesbezüglich gleich der aktuellen, aber mit geringerem Kommunikationsvolumen. Bei der Kommunikation wurde wiederum als erstes die Kommunikationsspitzen betrachtet, also das Maximum an auszutauschenden Daten zwischen zwei Prozessoren, und als weiteres, falls auch diese Größe bei beiden der zu vergleichenden Lösungen gleich war, die gesamte, zwischen alle Prozessoren zu bewältigenden Kommunikation. Entsprechend der Variante B 1 wurden auch die anderen Varianten des Kriteriums B modifiziert.

Bei dieser Testserie ergaben sich dieselben Verhältnisse hinsichtlich Qualität der Lösung und dem Aufwand zu ihrer Ermittlung wie in der erste Serie.

6. Das Zuordnungsverfahren bei MUPSI

Die Kenntnisse, die durch die kritische Auseinandersetzung mit den verschiedenen in der Literatur vorgeschlagenen Methoden für die Optimierung allgemein und speziell für die Zuordnungsproblematik bzw. das Scheduling gewonnenen wurden (siehe Kapitel 3.), sowie die Erfahrungen aus dem Vergleich der unterschiedlichen Varianten der Tauschverfahren (siehe Kapitel 4. und 5.) fanden eine Anwendung in dem Entwurf eines Lösungsweges für eine konkrete Problemstellung, die sich bei dem Projekt "MUPSI" ergab.

6.1. Anforderungen

Bei dem Projekt MUPSI handelt es sich um ein Multi-prozessorsystem zur Simulation nachrichtentechnischer Systeme, das beispielsweise zur Simulation eines umfangreichen, kontinuierlichen Systems bei der Nachrichtenübertragung in Echtzeit (bis 100kHz) eingesetzt werden soll. Das System besteht aus n homogenen Einzelprozessoren (Task-Processor, TP), einem Peripherie-Prozessor als Schnittstelle zur realen Welt und einem Steuerrechner (Control-Processor), die über einen Ringbus zu einem Ring zusammengeschlossen sind (Bild 10c). Jeder dieser Prozessoren ist ein von den anderen unabhängiger Rechner mit eigenem Speicher und eigener Kontrollogik. Über den Ringbus kann im Prinzip jeder der Prozessoren mit jedem anderen kommunizieren; der Botschaftenaustausch ist jedoch nur in einer Richtung möglich, so daß die Kommunikation benachbarter Prozessoren bevorzugt wird. Die eigentliche Echtzeit-Signalverarbeitung erfolgt dabei in den Einzelprozessoren, während der relativ langsame Steuerrechner nur dazu dient, den Ablauf der Simulation vorzubereiten und zu überwachen.

Ein nachrichtentechnisches System besteht aus verschiedenen Hardware-Komponenten (Modulatoren, Filter, Entzerrer, usw.), welche anliegende Eingangssignale zu entsprechenden

Ausgangssignalen verarbeiten. Simuliert man diese Hardware-Komponenten, so erhält man Programm-Module fester Länge (sogenannte Blöcke), die auf die einzelnen Prozessoren verteilt werden müssen.

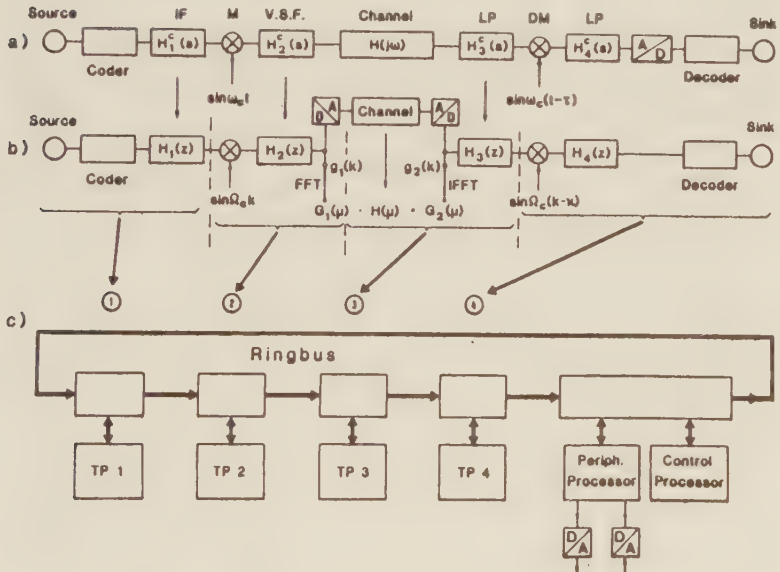


Bild 10: a) Ein Restseitenbandsystem zur Nachrichtenübertragung

b) Abbildung des Originalsystems in ein diskretes

c) Struktur des Multiprozessorsystems MUPSI

Bild 10a zeigt ein kleineres Beispiel für solch ein nachrichtentechnisches System, nämlich ein Restseitenbandsystem zur Nachrichtenübertragung. Als Vorbereitung für die eigentliche Simulation wird das Originalsystem vom Steuerrechner diskretisiert (Bild 10b). Anschließend erfolgt die Verteilung der verschiedenen Blöcke auf die Einzelprozessoren des Systems, beispielsweise wie in Bild 10c angedeutet.

Zur Nachbildung der im realen System bestehenden, fest verdrahteten Verbindungen wird der Ringbus verwendet. Eine analoge Verbindung der Blöcke untereinander wird erreicht, indem man die Ausgangssignale eines Blocks als "Datenpakete" über den Bus zu demjenigen Prozessor schickt, dem der Block zugeordnet wurde, der diese Daten eingabeseitig verarbeiten soll. Dieser Datenaustausch entfällt natürlich, wenn beide Blöcke dem gleichen Prozessor zugeteilt wurden. Eine derartige Datenverbindung wird als "Link" bezeichnet, wobei es der Fall sein kann, daß ein Link nicht nur einen, sondern mehrere Blöcke als Adressaten hat. Man erhält so ein Aufgabensystem, daß aus einer Reihe voneinander unabhängiger Blöcke besteht, deren einzige Verbindung die auszutauschenden Daten sind. Dies ist ein typisches Beispiel für ein Datenflußsystem. Die Hardware des Multiprozessorsystems ist speziell auf diese Art von Aufgabensystemen ausgelegt.

Die im Zusammenhang mit dieser Arbeit interessierende Aufgabe bei diesem Projekt ist es, die Blöcke so auf die Einzelprozessoren zu verteilen, daß alle in möglichst kurzer Zeit abgearbeitet werden können. Zeiten werden in Bearbeitungszyklen gemessen. Während eines Zyklusses kann, neben einem Rechenschritt pro Einzelprozessor, je eine Nachricht von einem Prozessor zum nächsten im Ring transportiert werden. Der Transport einer Nachricht zum übernächsten benötigt demzufolge zwei Zyklen usw.

Nun verarbeiten nachrichtentechnische Systeme natürlich nicht nur ein einziges Eingabezeichen, sondern einen ganzen Eingabestrom, der sich aus vielen, aufeinander folgenden

Zeichen zusammensetzt. Auf die Simulation übertragen bedeutet dies, daß nicht einmal n Blöcke zuzuordnen und abzuarbeiten sind, sondern daß ein Zuordnungsmuster zu erstellen ist, das periodisch immer wieder durchgerechnet wird.

Die einzelnen Zeichen des Eingabestroms werden im Pipelining-Betrieb verarbeitet, d. h. die einzelnen Blöcke bearbeiten in einem Intervall nicht das gleiche Zeichen, sondern Eingaben verschiedenen Alters. Nach einer Einschwingphase, die dem Durchlauf des ersten Zeichens entspricht, bearbeitet jeder Verarbeitungsblock unabhängig von den anderen das bei ihm als nächstes zur Eingabe anstehende Zeichen. Somit hat das System die angenehme Eigenschaft, daß keinerlei Reihenfolgen berücksichtigt werden müssen.

Wegen der periodischen Abarbeitung der Aufgaben ist es sinnvoll, sie speicherresident zu halten. Dies hat den Nachteil, daß es aufgrund der Gleichartigkeit der Einzelprozessoren zwar grundsätzlich möglich ist, jeden Block jedem Prozessor zuzuteilen. Bei der Zuordnung mehrerer Blöcke zu einem Prozessor, wie sich dies wegen der beschränkten Anzahl der Prozessoren (bis zu 11 sind vorgesehen) bei praktisch interessanten Simulationsbeispielen zwangsläufig durch deren Größe ergibt, muß beachtet werden, daß jeder Block Speicher in Anspruch nimmt, die Speicherkapazität eines Prozessors natürlich aber begrenzt ist.

Verschärfend kommt noch dazu, daß man es bei MUPSI mit drei getrennten Speichern pro Prozessor zu tun hat. Um die gewünschte Geschwindigkeit von 100 kHz Abtastfrequenz bei der Echtzeit-Simulation erreichen zu können, war es notwendig, daß schon der Einzelprozessor sehr schnell ist. Deshalb wurde im Rahmen des Projekts ein Prozessor entwickelt, dessen Funktionen hochgradig parallelisiert sind und den gleichzeitigen Zugriff zu drei Speichern gestatten, um so schneller alle für eine Operation benötigten Daten beschaffen zu können. Auch die Operationen im Prozessor laufen wieder pipeline-artig ab. Wegen dem Einsatz bei Echtzeit-Aufgaben handelt es sich bei den Speichern um Bau-

steine mit sehr schnellen Zugriffszeiten, die immer noch relativ teuer sind. Daher haben alle Speicher nur eine sehr begrenzte Kapazität. Es kann demzufolge möglich sein, daß eine Zuteilung, die die Blöcke so verteilt, daß die Rechenlast vollkommen gleich verteilt ist, wegen dieser Speicherrestriktionen nicht realisierbar und somit unzulässig ist.

Da das System so ausgelegt ist, daß der Ringbus kein Engpaß für die Übertragung der zur Kommunikation zwischen den Blöcken notwendigen Datenpakete sein kann, konnte auf eine Minimierung des Kommunikationsvolumens verzichtet werden. Ziel der Verteilung ist es also in diesem Fall nur, eine möglichst gleichmäßige Auslastung der Prozessoren zu erreichen unter Berücksichtigung der genannten Speicherrestriktionen. Dazu kommt die Besonderheit, daß gewisse Blöcke bei Bedarf zu diesem Zweck geteilt werden können auf Kosten eines gewissen Overheads an Speicherplatz- und Rechenzeitanforderungen.

Auf eine genauere Darstellung der Hardware des Multiprozessorsystems wurde an dieser Stelle verzichtet, da die über das schon geschilderte hinausgehenden Details im Zusammenhang mit der hier interessierenden Zuordnungsproblematik bedeutungslos sind. Der daran interessierte Leser sei auf die ausführliche Beschreibung des Systems in einem Artikel der Mitarbeiter dieses Projekts/Bol83a/ und auf die Kurzfassungen zweier Vorträge/Bol83b, Pol83a/ verwiesen.

6.2. Lösungsweg

6.2.1. Finden einer Erstlösung

Um für die skizzierte Aufgabenstellung eine geeignete, schon möglichst gute, erste Ausgangslösung für die anschließenden Tauschverfahren zu finden, wird hier als erstes eine Greedy-Methode eingesetzt. Es wird aus den noch nicht

zugewiesenen Blöcken derjenige ausgesucht, der die höchste Rechenzeitanforderung besitzt, und versucht, diesen dem Prozessor zuzuordnen, der die bisher geringste Rechenlast zu tragen hat. Paßt er wegen seiner Speicheranforderungen nicht mehr auf diesen, wird der Prozessor mit der nächsthöheren Rechenlast gewählt, usw. Diese Vorgehensweise entspricht der in Kapitel 3. beschriebenen Strategie LPTF.

Dieses Verfahren findet vielfach schon recht gute Lösungen, muß jedoch oft auch ohne Ergebnis abbrechen, d. h. ein Block kann keinem Prozessor zugeordnet werden. Dafür gibt es vor allem zwei Gründe. Erstens ist die Reihenfolge, in der die Blöcke zugeordnet werden, nicht in jedem Fall günstig. Und zweitens ist diese Strategie, wie es eben einem Greedy-Verfahren entspricht, nicht fähig, eine einmal vorgenommene Zuteilung wieder zurückzunehmen.

Daher wird bei ergebnislosem Abbruch von LPTF ein Rückverfolgungsalgorithmus ("backtracking"-Algorithmus) angewendet, der allerdings nur so lange systematisch Zuordnungen aufbaut, bis auch der letzte Block irgendeinem Prozessor zugeordnet, also eine erste zulässige Lösung gefunden wurde. Im Gegensatz zu LPTF wird hier nicht unbedingt eine gute Lösung gefunden. Es kann beispielsweise vorkommen, daß ein Prozessor gänzlich unberücksichtigt bleibt. Daher hat es sich als günstig erwiesen, diesem Algorithmus die Blöcke in einer Reihenfolge anzubieten, die sich an den Speicheranforderungen orientiert. Bricht auch dieses Verfahren ohne Resultat ab, ist die Aufgabenstellung unlösbar.

6.2.2. Verbesserung der Ausgangslösung

Ausgehend von der gefundenen Erstlösung wird die Zuordnung nun schrittweise verbessert, indem ein Block des Prozessors mit der augenblicklich höchsten Rechenlast gegen einen ausgetauscht wird, der auf einem anderen Prozessor liegt. Bei diesem Tausch verringert sich die Rechenlast des

ersteren, während die des zweiten nicht das bisherige Zeitmaximum erreichen darf.

Diese erste Tauschphase ist aber nicht in allen Fällen ausreichend. Dies hängt vor allem von der Erstlösung ab, von der ausgegangen wird. Mittels der Tauschoperationen wird zwar die Rechenlast verändert, die Zahl der zugeordneten Blöcke pro Prozessor bleibt jedoch hiervon unberührt. Es kann aber durchaus der Fall eintreten, daß von einer Ausgangslösung ausgegangen wird, bei der einem Prozessor gar kein Block zugeordnet wurde (insbesondere dann, wenn zur Ermittlung der Lösung der Rückverfolgungsalgorithmus eingesetzt werden mußte). An diesem Umstand wird auch durch die Tauschoperationen nichts geändert.

Daher wurde dieser Tauschphase noch eine Prozedur vorgelagert, die, solange es geht, Blöcke von dem höchstbelasteten Prozessor auf irgendeinen anderen schiebt - selbstverständlich ohne daß dabei die bisher erreichte Auslastung wieder verschlechtert wird.

Sind keine paarweisen Tauschoperationen mehr möglich, wird in der nächsten Phase versucht, für ein Paar von Prozessen des meistbelasteten Prozessors einen Tauschpartner zu finden. Entsprechend dreht sich das Verhältnis in der dritten Phase um, indem nun für einen Block des am stärksten ausgelasteten Prozessors ein Paar von Blöcken eines anderen Prozessors zum Tausch gesucht wird. Diese Phasen erwiesen sich als sinnvoll, da mittels des einfachen, paarweisen Tauschs oft für größere Blöcke kein Tauschpartner gefunden werden konnte. Ziel dieser beiden Phasen ist es daher, den Tausch eines "dicken" Blocks gegen zwei kleinere zu ermöglichen. Die Unterscheidung zwischen "zwei gegen einen" und "einer gegen zwei" ist hier deshalb nötig, da die Tauschalgorithmen die Blöcke immer vom belastesten Prozessor aus sehen und für die diesem zugewiesenen Blöcke Tauschpartner suchen.

Konnte in einer dieser drei Tauschphasen eine Verbesserung des Zielfunktionswertes erreicht werden, also die

Prozessoren etwas gleichmäßiger ausgelastet werden, beginnt ein neuer Zyklus von Verschieben und Tauschen von Blöcken, usw. bis sich keine Verbesserung mehr erzielen läßt.

Als letztes wird noch versucht, zwei Blöcke des einen Prozessors gegen zwei eines weniger belasteten zu tauschen. Ist dies möglich, wird der gesamte Zyklus des Verschiebens und der vier Tauschphasen erneut gestartet, bis auch in der letzten Phase keinerlei Verbesserungen mehr möglich sind.

Sicherlich gibt es auch Aufgabenstellungen, die durch Einführung zusätzlicher Prozeduren, die den Tausch noch größerer Tupel von Blöcken (wie z. B. zwei gegen drei, drei gegen drei, usw.) noch besser bearbeitet werden können. Jedoch steigt der Rechenaufwand dabei immer mehr an. Schon beim Tausch von zwei Blöcken gegen zwei andere ist er durchaus spürbar. Daher arbeitet diese Prozedur auch anders als die anderen. Während jene erst dann abbrechen, wenn keine entsprechende Tausch- oder Schiebeoperation mehr möglich ist, wird die letzte Tauschphase nach einer gewissen Anzahl an erfolgreichen Vertauschungen (z. Z. fünf) schon abgebrochen. Prozeduren, die gleichzeitig noch mehr Blöcke berücksichtigen, würden zuviel Zeit beanspruchen, ohne daß sich daraus spürbare Verbesserungen ergeben würden. Daher wurde auf eine Aufnahme weiterer Tauschvarianten in dieses Verfahren verzichtet.

6.2.3. Teilung von Blöcken

Es gibt verschiedene Gründe, weshalb die Teilung eines Blocks sinnvoll sein kann. Einer ergibt sich schon bei der Suche nach einer ersten Lösung.

Hat man im Aufgabensystem einen Block mit sehr hohen Speicheranforderungen, so kann es sein, daß keine Zuordnung ermittelt werden kann. Da es bei diesem konkreten Anwendungsfall tatsächlich solche teilbaren Blöcke gibt, wurde für diesen Fall vorgesehen, daß ein speicherintensiver

Block, wenn möglich, geteilt und daraufhin der Rückverfolgungsalgorithmus erneut aufgerufen wird. Falls dieser wiederum erfolglos terminiert, können weitere Teilungen vorgenommen werden.

Solche Teilungen haben jedoch nicht nur Vorteile. Ein teilbarer Block hat üblicherweise einen gewissen konstanten Bedarf an Speicherplatz für Daten in den drei Speichern. Dazu kommt ein bestimmter variabler Bedarf, der von einer blockspezifischen Größe abhängig ist. Dies kann z. B. bei einem Block, der ein Filter simuliert, der Grad dieses Filters sein. Außerdem muß in einem Speicher das Programm, welches den Block bearbeiten soll, abgelegt werden. Teilbar ist nun nur dieser variable Datenspeicherplatzbedarf. So entsteht durch jede Teilung ein gewisser Overhead an Speicherbedarf, da der konstante Bedarf an Datenspeicher nun mehrfach vorgesehen werden muß und auch das Programm selbst nun in zwei Prozessoren abgelegt werden muß.

Aber nicht nur der Speicherbedarf erhöht sich, sondern auch die insgesamt vom System zu erbringende Rechenzeit, da das Programm je einmal für jede Blockhälfte durchgerechnet werden muß. Damit erhöht sich auch zwangsweise die Anzahl an Zyklen bei der bestmöglichen Auslastung. Dies ist bei der Suche nach einer Erstlösung sicherlich noch nicht unbedingt von Belang, ja ist eine erste Ausgangslösung sonst überhaupt nicht möglich, wird man diesen Mehraufwand gerne in Kauf nehmen.

Mit den Teilungen hat man aber auch ein Instrument zur Verfügung, daß weitere Verbesserungen der durch die Tauschverfahren gefundenen Endlösung ermöglicht. In diesem Fall ist jedoch vorher abzuwägen, inwieweit sich der dadurch entstehende Overhead lohnen wird. Auf jeden Fall wird der Einsatz von Teilungen dort sinnvoll sein, wo ein sehr rechenintensiver Block eine weitere Verbesserung des Ziel-funktionswertes unmöglich macht.

Die Hälften des geteilten Blockes werden verschiedenen Prozessoren vorab zugeteilt und dann versucht, mittels der Tausch- und Verschiebungsverfahren die Lösung weiter zu verbessern. Ergibt es sich, daß z. B. aufgrund einer Vielzahl von Teilungen in der Endlösung doch wieder Teile desselben Ursprungsblocks (nicht notwendigerweise Hälften!) dem gleichen Prozessor zugeordnet sind, können diese wieder verschmolzen werden, da der Programmteil von der Teilung ja nicht berührt wurde, sondern nur der Datenteil. Somit kann in diesem Fall wieder die Rechenzeit und der Speicherplatz des durch die Teilung entstandenen Overheads eingespart werden. Dadurch können eventuell weitere Tauschoperationen ermöglicht werden, jedoch nicht viele, da es nur einen kleinen Unterschied zu überbrücken gilt.

Weitere Details des Verfahrens mit vielen Beispielen zur Illustration der Vorgehensweise können der Arbeit von Jarschel/Jar82a/ entnommen werden.

7. Schlußbemerkungen

Aus den in Kapitel 3 und 4 geschilderten und selbst gemachten Erfahrungen (Kapitel 5) ist zu schließen, daß die Tauschverfahren und die ihnen ähnlichen des direkten Suchens wohl mit zu den bedeutensten heuristischen Methoden zu zählen sind. Die Tatsache, daß heuristische Verfahren allgemein gesehen und Tauschverfahren insbesondere bis heute noch keine nennenswerte Verbreitung gefunden haben, ist wohl auf eine generell skeptische Einstellung gegenüber heuristisch gefundenen Lösungen zurückzuführen.

Da man aber für Probleme, die die Eigenschaft der NP-Vollständigkeit besitzen, nie einen praktisch sinnvoll einsetzbaren Algorithmus finden kann, und sich immer mehr heuristische Ansätze finden (wie z. B. die Tauschverfahren), die gute Lösungen bei günstigem Rechenzeitbedarf liefern, läßt sich doch eine zunehmende Bedeutung heuristischer Ansätze auch in den Bereichen erwarten, die heute oft noch von exakten Verfahren beherrscht sind. Gerade für die Zuordnungsproblematik scheinen zur Zeit noch überwiegend "Branch and Bound"-Verfahren im Einsatz zu sein, trotz des damit verbundenen nicht abschätzbaren Verhaltens bezüglich der Rechenzeit, wie dies für alle NP-vollständigen Algorithmen zutrifft. Umso beachtenswerter ist das gute Durchschnittsverhalten der Tauschverfahren bei diesem Problemkreis, von dem es kaum negative Abweichungen gibt.

Eine Forderung an einen guten Algorithmus war es auch, daß er sich nicht nur im Normalfall vernünftig verhalten soll, sondern daß auch die Wahrscheinlichkeit, daß er eine hinsichtlich des Optimierungskriteriums sehr schlechte Lösung liefert, möglichst gering sein soll. Auch diese Forderung wird von den Tauschverfahren erfüllt.

Es erscheint daher sinnvoll, bei der Suche nach geeigneten Verfahren doch vermehrt auch zu prüfen, ob Tauschverfahren oder die des direkten Suchens nicht auch für andere Problemkreise mit guten Resultaten einsetzbar sind.

Ein anderer Punkt für weitere Forschungsaktivitäten läßt sich in weiteren Untersuchungen darüber sehen, welche Kriterien für die Wahl einer geeigneten Nachbarschaft für eine Bezugslösung und der Auswahl einer Lösung daraus als nächsten Nachfolger sich noch als brauchbar zeigen.

Da das als Testhilfe entwickelte Dialogsystem modular aufgebaut ist, ließe es sich auch als Grundstock einer Verfahrensbibliothek für eingeweierte Benutzer verwenden. Außer den in Kapitel 5 genannten Methoden lassen sich jederzeit weitere Verfahren als eigenständige Module eingliedern, wobei lediglich das neue Kommando und das dann zu startende Programm in der Liste der verfügbaren Befehle des Dialogsteuerungsmoduls eingetragen werden müssen. Damit besteht für einen Benutzer, der die genaue Vorgehensweise und die Besonderheiten der einzelnen, zur Verfügung stehenden Verfahren kennt, die Möglichkeit, im Dialog verschiedene Varianten und Kombinationen der Methoden zu testen, um so für ein spezielles Problem ein geeignetes optimales, suboptimales oder heuristisches Verfahren herauszufinden.

Anhang

Exemplarisch seien hier die Daten einiger der Testbeispiele aufgeführt, auf die sich die zusammenfassende Darstellung der Ergebnisse von Kapitel 5 stützt. Dabei werden folgende Abkürzungen verwendet:

Abkürzungen:

DSP1	Datenspeicher 1
DSP2	Datenspeicher 2
PRSP	Programmspeicher
TIME	Zeitbedarf der Aufgabe (Einheit beliebig)
E1	Dateneingang 1
E2	Dateneingang 2
A1	Datenausgang 1
maxtime	maximal von einem Prozessor benötigte Zeit zur Fertigstellung aller ihm zugeordneten Aufgaben
opttime	rein rechnerische Optimalzeit bei totaler Gleichverteilung: Max (Summe der Rechenzeiten der Aufgaben durch Anzahl der Prozessoren, maximaler Zeitbedarf einer Aufgabe), aufgerundet auf ganze Recheneinheiten
procN	Prozessor N

Ohne Beschränkung der Allgemeinheit wurde allen Beispielen eine Hardware-Struktur zugrunde gelegt, die aus vier, im Ring zusammengeschalteten Prozessoren mit nachfolgendem Speicherausbau besteht:

Prozessordaten:

Nr.	maxDSP1	maxDSP2	maxPRSP
1:	4096	4096	4096
2:	4096	4096	4096
3:	4096	4096	4096
4:	4096	4096	4096

Die Zahlen geben dabei die maximal verfügbaren Speichereinheiten für die verschiedenen Speicher der Prozessoren an. Diese Einheiten können beliebig gedeutet werden, z. B. als Bytes oder Worte.

Die Datenpfade zwischen den Programmen sind durchnummeriert. Demzufolge ist das Ausgangsdatum mit der Nummer i Eingangsdatum bei allen denjenigen Aufgaben, die an einem Eingang ein Datum mit der Nummer i anliegen haben. Die Nummer 0 bedeutet, daß an diesem Eingang kein Datum erwartet wird.

Der in den Beispielen angegebene Zeitaufwand wird einerseits absolut, andererseits relativ angegeben. Die absolut zum Auffinden der Endlösung benötigte Zeit wurde auf einer Perkin Elmer 3210 gemessen. Die Angaben ist wie folgt aufgebaut, wobei die Stunden und Minuten nur aufgeführt werden, wo sie relevant sind, d. h. größer Null:

Stunden : Minuten : Sekunden . 10tel Sekunden

Die relative Zeitangabe bezieht sich auf die schnellste, optimale Lösung (wie auch schon in den Tabellen von Kapitel 5.). Diese beste Lösung bekommt den Wert 1, die anderen ein entsprechendes Vielfaches davon.

Testläufe, die mehr als 7.5 Stunden CPU-Zeit benötigten, wurden abgebrochen. In den Tabellen werden zum Teil die Daten der bis dahin gefundenen besten Lösung aufgeführt.

BEISPIEL 1:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	91	1024	100	70	1	2	0
2.	2086	800	100	90	2	3	0
3.	500	3000	100	100	3	4	0
4.	0	100	500	60	4	5	0
5.	239	239	468	239	5	6	0
6.	239	239	468	239	6	7	0
7.	0	15	780	190	7	8	0
8.	0	15	780	190	8	9	0
9.	50	50	68	70	9	10	0
10.	200	30	98	48	10	11	0
11.	20	68	140	30	11	12	0
12.	90	90	90	10	12	13	0

Erstlösung:

Prozessor 1 : 2 3 4 8 9 10 12

Prozessor 2 : 1 5 6 7 11

Prozessor 3 :

Prozessor 4 :

maxtime 768 opttime 334

maxtime/opttime 2.299

	proc1	proc2	proc3	proc4
Rechenlasten:	568	768	0	0
Speicherbelegung DSP1:	2926	589	0	0
Speicherbelegung DSP2:	4085	1585	0	0
Speicherbelegung PRSP:	1736	1956	0	0

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1, B1	339	1.2	1.00
A1, B2	339	2.3	1.92
A1, B3	339	3.6	3.00
A1, B4	339	3.2	2.67
A2, B1	380	1.2	1.00
A2, B2	339	1.7	1.42
A2, B3	339	1.8	1.50
A2, B4	339	1.9	1.58
A3, B1	339	1.7	1.42
A3, B2	339	2.3	1.92
A3, B3	380	2.0	1.67
A3, B4	339	2.6	2.17
A4, B1	339	1.7	1.42
A4, B2	339	2.1	1.75
A4, B3	339	1.3	1.92
A4, B4	339	2.3	1.92
backtrack	339	9:54.4	99.50

BEISPIEL 2:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	91	1023	100	70	1	2	0
2.	2086	800	100	72	2	3	0
3.	1000	1096	100	97	3	4	0
4.	596	100	100	83	4	5	0
5.	96	25	100	52	5	6	0
6.	96	400	30	47	6	7	0
7.	2096	600	196	51	7	8	0
8.	96	100	196	48	8	9	0
9.	1096	50	596	49	9	10	0
10.	100	24	50	52	10	11	0
11.	596	60	100	87	11	12	0
12.	119	200	200	40	12	13	0
13.	60	300	30	100	13	14	0
14.	400	40	40	50	14	15	0
15.	70	20	12	60	15	16	0
16.	60	96	70	60	16	17	0
17.	80	96	30	50	17	18	0
18.	20	596	80	30	18	19	0
19.	60	100	96	40	19	20	0
20.	50	70	50	50	19	21	20

Erstlösung:

Prozessor 1 : 1 2 3 4 5 6 10 18
Prozessor 2 : 7 8 9 11 12 13
Prozessor 3 : 14 15 16 17 19 20
Prozessor 4 :

maxtime 503 opttime 297
maxtime/opttime 1.694

	proc1	proc2	proc3	proc4
Rechenlasten:	503	375	310	0
Speicherbelegung DSP1:	4085	4063	720	0
Speicherbelegung DSP2:	4064	1310	422	0
Speicherbelegung PRSP:	660	1318	298	0

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1,B1	298	2.6	0.93
A1,B2	297	9.0	3.21
A1,B3	297	20.4	7.29
A1,B4	298	10.6	3.79
A2,B1	298	2.6	0.93
A2,B2	297	4.8	1.71
A2,B3	297	4.9	1.75
A2,B4	298	4.3	1.54
A3,B1	297	3.5	1.25
A3,B2	297	4.6	1.64
A3,B3	298	7.3	2.61
A3,B4	297	4.8	1.71
A4,B1	297	2.8	1.00
A4,B2	297	5.2	1.86
A4,B3	297	5.8	2.07
A4,B4	298	5.5	1.96
backtrack	299	> 7.5 Stunden	

BEISPIEL 3:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	91	1023	100	70	1	2	0
2.	2086	800	100	72	2	3	0
3.	1000	1096	100	97	3	4	0
4.	596	100	100	83	4	5	0
5.	96	25	100	52	5	6	0
6.	96	400	30	47	6	7	0
7.	2096	600	196	51	7	8	0
8.	96	100	196	48	8	9	0
9.	1096	50	596	49	9	10	0
10.	100	24	50	52	10	11	0
11.	596	60	100	87	11	12	0
12.	119	200	200	40	12	13	0
13.	60	300	30	100	13	14	0
14.	400	40	40	50	14	15	0
15.	70	20	12	60	15	16	0
16.	60	96	70	60	16	17	0
17.	80	96	30	50	17	18	0
18.	20	596	80	30	18	19	0
19.	60	100	96	40	19	20	0
20.	50	70	50	50	19	21	20
21.	60	10	596	100	21	22	0
22.	30	596	1000	30	22	23	0
23.	800	500	10	60	23	24	0
24.	100	300	30	50	24	25	0
25.	100	200	200	50	25	26	0
26.	96	400	30	60	26	27	0
27.	100	400	490	30	27	28	0
28.	100	60	10	70	28	29	0
29.	100	60	500	70	29	30	0
30.	100	300	500	60	30	31	0

Erstlösung:

Prozessor 1 : 1 2 3 21 22 23
 Prozessor 2 : 4 7 9 13 18 27 30
 Prozessor 3 : 5 6 8 10 11 12 14 15 16 17 19 20 24 25 26 28 29
 Prozessor 4 :

maxtime 936 opttime 442
 maxtime/opttime 2.118

	proc1	proc2	proc3	proc4
Rechenlasten:	429	403	936	0
Speicherbelegung DSP1:	4067	4068	2319	0
Speicherbelegung DSP2:	4025	2346	2251	0
Speicherbelegung PRSP:	1906	1992	1744	0

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1,B1	442	4.1	1.05
A1,B2	444	27.5	7.05
A1,B3	443	3:53.1	59.77
A1,B4	442	46.2	11.85
A2,B1	442	3.9	1.00
A2,B2	443	17.9	4.59
A2,B3	443	21.9	5.62
A2,B4	443	17.6	4.51
A3,B1	442	5.5	1.41
A3,B2	443	18.6	4.77
A3,B3	443	25.8	6.62
A3,B4	443	18.9	4.85
A4,B1	443	7.5	1.92
A4,B2	443	18.2	4.67
A4,B3	443	25.8	6.62
A4,B4	442	23.2	5.95
backtrack		> 7.5 Stunden	

BEISPIEL 4:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	91	1023	100	70	1	2	0
2.	2086	800	100	72	2	3	0
3.	1000	1096	100	97	3	4	0
4.	596	100	100	83	4	5	0
5.	96	25	100	52	5	6	0
6.	96	400	30	47	6	7	0
7.	2096	600	196	51	7	8	0
8.	96	100	196	48	8	9	0
9.	1096	50	596	49	9	10	0
10.	100	24	50	52	10	11	0
11.	596	60	100	87	11	12	0
12.	119	200	200	40	12	13	0
13.	60	300	30	100	13	14	0
14.	400	40	40	50	14	15	0
15.	70	20	12	60	15	16	0
16.	60	96	70	60	16	17	0
17.	80	96	30	50	17	18	0
18.	20	596	80	30	18	19	0
19.	60	100	96	40	19	20	0
20.	50	70	50	50	19	21	20
21.	60	10	596	100	21	22	0
22.	30	596	1000	30	22	23	0
23.	800	500	10	60	23	24	0
24.	100	300	30	50	24	25	0
25.	100	200	200	50	25	26	0
26.	96	400	30	60	26	27	0
27.	100	400	490	30	27	28	0
28.	100	60	10	70	28	29	0
29.	100	60	500	70	29	30	0
30.	100	300	500	60	30	31	0

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
31.	40	60	400	90	31	32	0
32.	1000	50	1000	70	32	33	0
33.	200	30	40	60	33	34	0
34.	50	50	500	50	34	35	0
35.	30	400	300	40	35	36	0
36.	40	500	400	40	36	37	0
37.	40	30	40	20	37	38	0
38.	20	500	500	30	38	39	0
39.	20	500	700	30	39	40	0
40.	40	500	300	20	40	41	0
41.	40	500	30	30	41	42	0
42.	40	400	500	60	42	43	0
43.	30	500	800	40	43	44	0
44.	40	500	30	39	44	45	0
45.	20	10	10	38	45	46	0
46.	30	200	300	39	46	47	0
47.	20	300	300	40	47	48	0
48.	30	300	300	50	48	49	0
49.	30	700	500	36	49	50	0
50.	20	190	500	39	50	51	0

Erstlösung:

```

Prozessor  1 :  1  3  7 21 22 23
Prozessor  2 :  2  4  9 18 29 30 39 43 49
Prozessor  3 :  6 11 14 26 32 34 36 38 40 41 42 44 50
Prozessor  4 :  5  8 10 12 13 15 16 17 19 20 24 25 27 28 31 33
                35 37 45 46 47 48

```

```

maxtime 1129          opttime  658
maxtime/opttime 1.716

```

	proc1	proc2	proc3	proc4
Rechenlasten:	408	470	622	1129
Speicherbelegung DSP1:	4077	4078	2478	1601
Speicherbelegung DSP2:	3825	3606	4090	3321
Speicherbelegung PRSP:	2002	3876	3960	3254

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1,B1	658	18.8	1.00
A1,B2	658	1:11:00.0	226.60
A1,B3		> 7.5 Stunden	
A1,B4	658	2:59:16.7	575.16
A2,B1	658	24.4	1.30
A2,B2	658	31.7	1.69
A2,B3	658	23.3	1.24
A2,B4	658	34.6	1.84
A3,B1	658	23.6	1.26
A3,B2	658	32.2	1.71
A3,B3	658	40.8	2.17
A3,B4	658	32.7	1.74
A4,B1	658	21.3	1.13
A4,B2	658	33.0	1.76
A4,B3	658	46.0	2.45
A4,B4	659	51.7	2.75
backtrack		> 7.5 Stunden	

BEISPIEL 5:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	0	21	50	47	1	2	0
2.	32	80	50	195	2	3	0
3.	2	24	50	52	3	4	0
4.	32	80	50	195	4	5	0
5.	4096	1024	200	56	5	6	0
6.	4096	1024	200	56	6	7	0
7.	32	80	50	195	7	8	0
8.	2	24	50	52	8	9	0
9.	32	80	50	195	9	10	0
10.	0	21	50	47	10	11	0

Erstlösung:

Prozessor 1 : 1 5 10

Prozessor 2 : 6

Prozessor 3 : 2 3 4 7 8 9

Prozessor 4 :

maxtime 884 opttime 273

maxtime/opttime 3.238

	proc1	proc2	proc3	proc4
Rechenlasten:	150	56	884	0
Speicherbelegung DSP1:	4096	4096	132	0
Speicherbelegung DSP2:	1066	1024	368	0
Speicherbelegung PRSP:	300	200	300	0

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1,B1	442	1.1	1.00
A1,B2	442	1.5	1.36
A1,B3	442	1.5	1.36
A1,B4	442	1.3	1.18
A2,B1	442	1.2	1.09
A2,B2	442	1.3	1.18
A2,B3	442	1.4	1.27
A2,B4	442	1.4	1.27
A3,B1	442	1.3	1.18
A3,B2	442	1.4	1.27
A3,B3	442	1.6	1.45
A3,B4	442	1.5	1.36
A4,B1	442	1.2	1.09
A4,B2	442	1.4	1.27
A4,B3	442	1.5	1.36
A4,B4	442	1.5	1.36
backtrack	442	4.2	3.82

BEISPIEL 6:

Daten der Aufgaben:

NR	DSP1	DSP2	PRSP	TIME	E1	A1	E2
1.	0	21	50	47	1	2	0
2.	32	80	50	195	2	3	0
3.	2	24	50	52	3	4	0
4.	32	80	50	195	4	5	0
5.	2073	537	200	53	5	6	0
6.	2073	537	200	53	6	7	0
7.	2073	537	200	53	7	8	0
8.	2073	537	200	53	8	9	0
9.	32	80	50	195	9	10	0
10.	2	24	50	52	10	11	0
11.	32	80	50	195	11	12	0
12.	0	21	50	47	12	13	0

Erstlösung:

Prozessor 1 : 1 5 12

Prozessor 2 : 6

Prozessor 3 : 2 3 4 7 9 10 11

Prozessor 4 : 8

maxtime 937 opttime 298

maxtime/opttime 3.144

	proc1	proc2	proc3	proc4
Rechenlasten:	147	53	937	53
Speicherbelegung DSP1:	2073	2073	2205	2073
Speicherbelegung DSP2:	579	537	905	537
Speicherbelegung PRSP:	300	200	500	200

Ergebnisse:

Variante	maxtime	Zeitaufwand	
		absolut	relativ
A1,B1	300	1.2	1.09
A1,B2	300	2.4	2.18
A1,B3	300	2.9	2.64
A1,B4	300	2.8	2.55
A2,B1	300	1.1	1.00
A2,B2	300	1.4	1.27
A2,B3	300	1.5	1.36
A2,B4	300	1.6	1.45
A3,B1	300	1.5	1.36
A3,B2	300	1.8	1.64
A3,B3	300	2.0	1.81
A3,B4	300	2.0	1.81
A4,B1	300	1.4	1.27
A4,B2	300	2.3	2.09
A4,B3	300	1.7	1.55
A4,B4	300	1.9	1.73
backtrack	300	2:09.4	117.64

Literaturverzeichnis

- Aro80a. Arora, R.K. and Rana, S.P., "Heuristic Algorithms for Process Assignment in Distributed Computing Systems," Information Processing Letters Vol. 11(4,5), pp.199-203 (12 December 1980).
- Bal65a. Balas, E., "An Additive Algorithm for Solving Linear Programs with Zero-One Variables," Operations Research Vol. 13(4), pp.517-546 (1965).
- Bal67a. Balas, E., "Discrete Programming by the Filter Method," Operations Research Vol. 15, pp.915-957 (1967).
- Bal79a. Balas, E. and Guignard, M., "Report of the Session on Branch and Bound / Implicit Enumeration," Annals of Discrete Mathematics Vol. 5, pp.185-191 (1979).
- Bal70a. Balinski, M.L., "Integer Programming: Methods, Uses, Computation," pp. 199-266 in Proceedings of the Princeton Symposium on Mathematical Programming, ed. H.W. Kuhn, Princeton University Press, Princeton, New Jersey (1970).
- Bok81a. Bokhari, S.D., "On the Mapping Problem," IEEE Transactions on Computers Vol. C-30(3), pp.207-214 (March 1981).
- Bol83b. Bolch, G., Hofmann, F., Hoppe, B., Linster, C.U., and Wurm, F.X., "A Multiprocessor Structure for Signal Processing," pp. 805-807 in Signal Processing II: Theories and Applications, ed. W. Schüßler, Elsevier Science Publishers B/V. (North-Holland), Amsterdam (1983). Proceedings of EUSIPCO-83
- Bol83a. Bolch, G., Hofmann, F., Hoppe, B., Kolb, H.J., Linster, C.U., Polzer, R., Schüßler, W., Wackersreuther, G., and Wurm, F.X., "A Multiprocessor System for Simulating Data Transmission Systems (MUPSI),"

- Microprocessing and Microprogramming (The EUROMICRO Journal) Vol. 12(5), pp.267-277 (December 1983).
- Bär76a. Bär, G., "Zur Linearisierung beliebiger 0-1-Optimierungsprobleme," Math. Operationsforsch. u. Statist. Vol. 7(2), pp.181-195 (1976).
- Cam70a. Campbell, H.G., Dudek, R.A., and Smith, M.L., "A Heuristic Algorithm for the n Job, m Machine Sequencing Problem," Management Science Vol. 16(10), pp.B630-B637 (June 1970).
- Cha70a. Charlton, J.M. and Death, C.C., "A Method of Solution for General Machine-Scheduling Problems," Operations Research Vol. 18, pp.689-707 (1970).
- Chu80a. Chu, W.W., Holloway, L.J., Lan, M.T., and Efe, K., "Task Allocation in Distributed Data Processing," Computer Vol. 13, pp.57-69 (November 1980).
- Cof73a. Coffman, E. and Denning, P., Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
- Dak65a. Dakin, R.J., "A Tree Search Algorithm for Mixed Integer Programming Problems," The Computer Journal Vol. 8, pp.250-255 (1965).
- Dan66a. Dantzig, G.B., Lineare Programmierung und Erweiterungen, Springer Verlag, Berlin-Heidelberg-New York (1966).
- Düc77a. Dück, W., Diskrete Optimierung, Vieweg, Braunschweig (1977). Reihe Wissenschaft
- Ech68a. Echols, R.E. and Cooper, L., "Solution of Integer Linear Programming Problems by Direct Search," Journal of ACM Vol. 15(1), pp.75-84 (1968).

- Eck77a. Ecker, K., Organisation von parallelen Prozessen: Theorie deterministischer Schedules, Bibliographisches Institut, Mannheim-Wien-Zürich (1977). Reihe Informatik; Bd. 23
- For62a. Ford, L.R. and Fulkerson, D.R., Flows in Networks, Princeton University Press, Princeton, N.J. (1962).
- Fre82a. French, S., Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop, Ellis Horwood Ltd., Chichester (1982).
- Fre81a. Freundl, M., Multiprozessorprojekt MUPSI: Untersuchung von Strategien zur Verteilung der Anwenderprozesse auf die Einzelprozessoren, Universität Erlangen-Nürnberg, IMMD IV, Erlangen (1981). Diplomarbeit
- Fro82a. Fromm, H., "Multiprozessor-Rechenanlagen: Programmstrukturen, Maschinenstrukturen und Zuordnungsprobleme," Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg Vol. 15(5), Dissertation (Mai 1982).
- Gal76a. Gallus, G., "Heuristische Verfahren zur Lösung allgemeiner ganzzahliger linearer Optimierungsprobleme (Ein Überblick)," Operations Research Vol. 20, pp.89-104 (1976).
- Gar78a. Garey, M.R., Graham, R.L., and Johnson, D.S., "Performance Guarantees for Scheduling Algorithms," Operations Research Vol. 26(1), pp.3-21 (1978).
- Gar79a. Garey, M.R. and Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco (1979).
- Gar72a. Garfinkel, R.S. and Nemhauser, G.L., Integer Programming, John Wiley & Sons, New York-London (1972).

- Geo72a. Geoffrion, A.M and Marsten, R.E., "Integer Programming Algorithms: A Framework and State-of-Art Survey," Management Science Vol. 18(9), pp.465-491 (May 1972).
- Geo76a. Geoffrion, A.M., "A Guided Tour of Recent Practical Advances in Integer Linear Programming," Omega Vol. 4(1), pp.49-57 (1976).
- Glo65a. Glover, F., "A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem," Operations Research Vol. 13, pp.879-919 (1965).
- Gom58a. Gomory, R.E., An Algorithm for Integer Solutions to Linear Programs, IBM Math. Res. Rep., Princeton (Nov. 1958).
- Gre68a. Greenberg, H.H., "A Branch-Bound Solution to the General Scheduling Problem," Operations Research Vol. 16, pp.353-361 (1968).
- Hae80a. Haessig, K. and Jenny, C.J., An Algorithm for Allocating Computational Objects in Distributed Computing, IBM Research Division, San Jose-Yorktown-Zurich (1980). Communications/Computer Science - Research Report
- Hae80b. Haessig, K. and Jenny, C.J., "Partitioning and Allocating Computational Objects in Distributed Computing Systems," pp. 593-598 in Proceedings of IFIP Congress, ed. S. Lavington, North Holland Publishing Company, Amsterdam-New York-Oxford (1980).
- Ham79a. Hammer, P.L., "Boolean Elements in Combinatorial Optimization," Annals of Discrete Mathematics Vol. 4, pp.51-71 (1979).
- Ham74a. Hammer, P.L., "Boolean Procedures for Bivalent Programming," pp. 311-363 in Mathematical Programming in Theory and Practice, ed. P.L. Hammer and G. Zoutendijk, North-Holland Publishing Company, Amsterdam-Oxford (1974).

- Ham68a. Hammer, P.L. and Rudeanu, S., Boolean Methods in Operations Research and Related Areas, Springer-Verlag, Berlin-Heidelberg-New York (1968).
- Han79a. Hansen, P., "Methods of Nonlinear 0-1 Programming," Annals of Discrete Mathematics Vol. 5, pp.53-70 (1979).
- Han69a. Hansen, P., "Note sur l'extension de la méthode d'énumération implicite aux programmes non linéaires en variables zéro-un," Cahiers du Centre d'Etudes de Recherche Opérationnelle Vol. 11, pp.162-166 (1969).
- Hil69a. Hillier, F.S., "Efficient Heuristic Procedures for Integer Linear Programming with an Interior," Operations Research Vol. 17(4), pp.600-637 (1969).
- Hor81a. Horowitz, E. and Sahni, S., Algorithmen, Springer-Verlag, Berlin-Heidelberg-New York (1981).
- Jar82a. Jarschel, W., Multiprozessorprojekt MUPSI: Entwurf und Implementierung des Verteilers, Universität Erlangen-Nürnberg, IMMD IV, Erlangen (1982). Diplomarbeit
- Kar72a. Karp, R.M., "Reducibility among Combinatorial Problems," pp. 85-103 in Complexity of Computer Computations, ed. J.W. Thatcher, Plenum Press, New York (1972).
- Kle80a. Klee, V., "Combinatorial Optimization: What is the State of the Art," Mathematics of Operations Research Vol. 5, pp.1-26 (February 1980).
- Kre70a. Kreuzberger, H., "Numerische Erfahrungen mit einem heuristischen Verfahren zur Lösung ganzzahliger linearer Optimierungsprobleme," Elektronische Datenverarbeitung Vol. 12(7), pp.289-306 (1970).

- Kre69a. Kreuzberger, H., Verfahren zur Lösung ganzzahliger linearer Optimierungsprobleme, Fakultät für Mathematik und Physik, Technische Hochschule Darmstadt, Darmstadt (1969). Dissertation
- Lag81a. Lageweg, B.J., Lawler, E.L., Lenstra, J.K., and Rinnooy Kan, A.H.G., Computer Aided Complexity Classification of Deterministic Scheduling Problems, Mathematisch Centrum, Amsterdam (1981). Report BW 138
- Law83a. Lawler, E.L., "Recent Results in the Theory of Machine Scheduling," pp. 202-234 in Mathematical Programming - The State of Art - Bonn 1982, ed. B. Korte, Springer Verlag, Berlin-Heidelberg-New York-Tokyo (1983).
- Law66a. Lawler, E.L. and Wood, D.E., "Branch-and-Bound Methods: A Survey," Operations Research Vol. 14, pp.699-719 (1966).
- Mer78a. Mertens, P., Industrielle Datenverarbeitung, Betriebswirtschaftlicher Verlag Dr. Th. Gabler, Wiesbaden (1978). Bd.1: Administrations- und Dispositionssysteme
- Mey73a. Meyer, M., Hansen, K., and Rohde, M., Mathematische Planungsverfahren I - Eine anwendungsorientierte Einführung in die Lineare Programmierung und die Netzplantechnik, Verlag W. Girardet, Essen (1973). Girardet-Taschenbücher Technik, Bd. 12
- Mül69a. Müller-Merbach, H., "Das Verfahren der 'vorsichtigen Annäherung' - Eine heuristische Methode zur Lösung gewisser Probleme der ganzzahligen Planungsrechnung," Elektronische Datenverarbeitung Vol. 11(12), pp.462-466 (1969).
- Mül75a. Müller-Merbach, H., "Modelling Techniques and Heuristics for Combinatorial Problems," pp. 3-27 in Combinatorial Programming: Methods and Applications, ed. B. Roy, Dordrecht (1975). Nato Advanced Study

Institutes Series, Series C, Volume 19

- Pet76a. Petrich, F. and Wagner, K., "Ein Algorithmus für das lineare 0-1-Optimierungsproblem," Math. Operationsforsch. u. Statist. Vol. 7(3), pp.351-359 (1976).
- Pie68a. Pierce, J.F., "Application of Combinatorial Programming to a Class of All-Zero-One Integer Programming Problems," Management Science Vol. 15(3), pp.191-209 (November 1968).
- Pol83a. Polzer, R. and Wackersreuther, G., "A Floating Point Signal Processor for Real Time Applications," pp. 801-804 in Signal Processing II: Theories and Applications, ed. W. Schüßler, Elsevier Science Publishers B/V. (North-Holland), Amsterdam (1983). Proceedings of EUSIPCO-83
- Rao79a. Rao, G.S., Stone, H.S., and Hu, T.C., "Assignment of Tasks in a Distributed Processor System with Limited Memory," IEEE Transactions in Computers Vol. C-28(4), pp.291-299 (April 1979).
- Rei65a. Reiter, S. and Sherman, G., "Discrete Optimizing," Journal of SIAM Vol. 13(3), pp.864-889 (1965).
- Sch78a. Schoch, M. and Lyska, W., "Ein weiterer kombinatorischer Algorithmus für 0-1-Optimierungsaufgaben," Math. Operationsforsch. u. Statist. Vol. 9, pp.307-319 (1978).
- Sch78b. Schoch, M. and Lyska, W., "Kombinatorische Algorithmen zur Lösung spezieller nichtlinearer 0-1-Optimierungsaufgaben," Math. Operationsforsch. u. Statist. Vol. 9(1), pp.9-20, Ser. Optimization (1978).
- Soc83a. Soceanu, A., Zuordnung von Rechenprozessen in verteilten Mikrocomputer-Systemen, HochschulVerlag, Freiburg (1983). HochschulSammlung Naturwissenschaft; Informatik; Bd. 4. Dissertation

- Sto78a. Stone, H.S. and Bokhari, S.H., "Control of Distributed Processes," Computer Vol. 11(7), pp.97-106 (July 1978).
- Sto78b. Stone, H.S., "Critical Load Factors in Two-Processor Distributed Systems," IEEE Transactions on Software Engineering Vol. SE-4(3), pp.254-258 (May 1978).
- Sto77a. Stone, H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Transactions on Software Engineering Vol. SE-3(1), pp.85-93 (January 1977).
- Wu80a. Wu, S.B. and Liu, M.T., "Assignment of Tasks and Resources for Distributed Processing," pp. 655-662 in Proc. Compcon 80 Fall (September 1980).
- Wu80b. Wu, S.B. and Liu, M.T., "A Partition Algorithm for Parallel and Distributed Processing," pp. 254-255 in Proceedings of the First International Conference on Parallel Processing (1980).
- Wun79a. Wunderatsch, H., "Deterministisches Zuordnen in heterogenen Prozessorsystemen mit Pseudo-Boole Methoden," Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg Vol. 12(2), Dissertation (Juni 1979).
- Zio72a. Zions, S., "Generalized Implicit Enumeration Using Bounds on Variables for Solving Linear Programs with Zero-One Variables," Naval Res. Logist. Quart. Vol. 19, pp.165-181 (1972).

Lebenslauf

Beate Hoppe wurde am 5.2.1955 in Hamburg geboren. Nach dem Abitur im Jahr 1973 am Katharineum zu Lübeck ließ sie sich bei der Firma Siemens in München zur Ingenieurassistentin der Fachrichtung Nachrichtentechnik ausbilden. Im Anschluß an diese zwei-jährige Ausbildung wurde sie von Siemens in Braunschweig übernommen, wo sie an Projekten im In- und Ausland zur Automatisierung von Rangierbahnhöfen mitarbeitete.

Im Wintersemester 1976/77 begann sie das Studium der Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg, das sie im Mai 1981 mit dem Diplom abschloß. Der Schwerpunkt ihres Studiums lag im Bereich der Betriebssysteme; als Nebenfach wählte sie Betriebswirtschaft.

Seit Juni 1981 ist sie Mitarbeiter am Lehrstuhl Informatik IV (Betriebssysteme) der Universität Erlangen im Rahmen des Projekts MUPSI, das von der Deutschen Forschungsgemeinschaft gefördert wird. Ihre Aufgabe ist das Portieren des Betriebssystems UNIX* auf den Steuerrechner für das Multiprozessorsystem, Erweiterung und Anpassung desselben an die dortigen Bedürfnisse und Wartung des Betriebssystems. Daneben gilt es verschiedene Werkzeuge für den künftigen Anwender des Multiprozessorsystems zu erstellen, um ihm ein problemorientiertes Arbeiten zu ermöglichen.

*) UNIX ist ein eingetragenes Warenzeichen von Bell Lab.

In letzter Zeit sind folgende Arbeitsberichte des Instituts für
Mathematische Maschinen und Datenverarbeitung erschienen:

Band 13

- =====
- Nr. 1 Schrammel, D., Bayer, W., Gliemann, H.-P., Heigl, M.,
Mertens, P.:
Ein Leistungs- und Wirtschaftlichkeitsvergleich zwischen
Klein- und Großcomputern
(Januar 1980)
- Nr. 2 Göttler, H.:
Semantische Modelle
(März 1980)
- Nr. 3 Fromm, H.J.:
Zur Modellierung der Speicherinterferenz bei hierarchisch
organisierten Multiprozessorsystemen
(April 1980)
- Nr. 4 Jäpel, D.:
Klassifikatorbezogene Merkmalsauswahl
(Juli 1980)
- Nr. 5 Bodendorf, F., Mertens, P., Haas, R., Weber, G., Wolf, G.:
SAMBA - ein Methodenbankrahmen um das Statistik-Paket SPSS
(Juli 1980)
- Nr. 6 Bartsch, B.:
Inferenz und Analyse spezieller Graphgrammatiken für die
syntaktische Mustererkennung
(September 1980)
- Nr. 7 Wurm, F.X.:
Auftragssystem für eine Mehrprozessoranlage
(Oktober 1980)
- Nr. 8 Schedelbeck, A.:
Vergleich der beiden höheren Realzeitprogrammiersprachen
ADA und PEARL
(Oktober 1980)
- Nr. 9 Jahresbericht 1980 der Informatik
(März 1981)

- Nr. 1 Pattern Recognition
 (Research at Lehrstuhl für Informatik 5)
 (Februar 1981)
- Nr. 2 Bunke, H.:
 Analyse elektrischer Schaltpläne
 (März 1981)
- Nr. 3 Linster, C.-U.:
 SYMPOS/UNIX - Ein Betriebssystem für homogene
 Polyprozessorsysteme
 (Juni 1981)
- Nr. 4 Akyildiz, J.F., Bolch, G.:
 Analytic Solution Techniques for Queueing Network
 Models of Computer Systems
 (Juli 1981)
- Nr. 5 Lindstedt, W.G.:
 Ein Verfahren zur Erstellung portabler algorithmischer
 Laufzeitprogramme
 (Juli 1981)
- Nr. 6 Vollmar, R.:
 Zum Begriff des Parallelismus bei Polyautomaten
 (Juli 1981)
- Nr. 7 Luft, A.L.:
 Rationaler Sprachgebrauch und orthosprachliche
 Standardisierung als Grundlagen zuverlässiger Software-
 Entwicklung und -Dokumentation
 (Januar 1982)
- Nr. 8 Händler, W.:
 PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
 ARCHITECTURE
 (Februar 1982)
- Nr. 9 Beth, Th., Hess, P., Wirl, K.:
 Materialien zur Kryptographie
 (März 1982)
- Nr. 10 Jahresbericht 1981 der Informatik
 (März 1982)

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hohl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)
- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr. 10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)

- Nr. 11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr. 12 Brendel, W., Fan, Z., Klar, R., Schmielau, W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr. 13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozeß-Programmiersprachen
(Oktober 1982)
- Nr. 14 Sommer, U., Steinchen, J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr. 15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)
- Nr. 16 Bolch, G., Bruchner, W.:
Analytische Modelle für symmetrische Mehrprozessor-
anlagen mit dynamischen Prioritäten
(Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
(März 1983)

- Nr. 1 Gall, R.:
Formale Beschreibung des inkrementellen Programmierens-
im-Großen mit Graph-Grammatiken
(März 1983)
- Nr. 2 König, R.:
Beiträge zur algebraischen Theorie der formalen Sprachen
(März 1983)
- Nr. 3 Welbourn, D.B.:
The Design of Mechanical Components and the Development
of DUCT:
17 Years of CAD/CAM in Cambridge University
(April 1983)
- Nr. 4 Woitok, R.:
Die System-Implementierungs-Sprache ALICE
(Mai 1983)
- Nr. 5 Schneider, S., Schwab, P., Renninger, W.:
Wesen, Vergleich und Stand von Software zur Produktion
von Systemen der computergestützten Unternehmensplanung
(Juli 1983)
- Nr. 6 Händler, W., Herzog, U., Hofmann, F., Schneider, H.J.
und Mitarbeiter:
Projektabschlußbericht 1978 - 1982 Erlangen General Purpose
Array
(Juli 1983)
- Nr. 7 Mackert, L.:
Modellierung, Spezifikation und korrekte Realisierung
von asynchronen Systemen
(Juli 1983)
- Nr. 8 Reitenspieß, M.:
Sprachkonstrukte zur Spezifikation und korrekten Implemen-
tierung von Schutzproblemen
(Juli 1983)
- Nr. 9 Weber, K.:
Modellierung von Fehlverhalten mit Berücksichtigung
paralleler Abläufe
(August 1983)
- Nr. 10 Dietsch, H.:
Ein verteiltes Kommunikationssystem für einen Verbund
aus lokalen Mikrorechner-Netzwerken
(Dezember 1983)

- Nr. 11 Maehle, E., Schmitter, E.:
Workshop Fehlertolerante Mehrprozessor- und Mehr-
rechnersysteme
(Dezember 1983)
- Nr. 12 Volkert, J.:
Multiplikative Abstandsfunktionen und deren Anwendung
in der Bildverarbeitung
(Dezember 1983)
- Nr. 13 Steinbauer, D.:
Transaktionen als Grundlage zur Strukturierung und
Integritätssicherung in Datenbank-Anwendungssystemen
(Dezember 1983)
- Nr. 14 Wilke, P.:
Zusammenhänge und Unterschiede zwischen Graph-Grammatiken
und Petri-Netzen sowie verwandter Systeme
(Dezember 1983)
- Nr. 15 Jahresbericht der Informatik 1983
(März 1984)

- Nr. 1 Eberlein, W.:
Architektur technischer Datenbanken für
integrierte Ingenieursysteme
(Februar 1984)
- Nr. 2 Mertens, P., Heigl, M.:
Neuere Entwicklungen der computergestützten
Produktionsplanung, Eignung - Verbindungen -
Entwicklungspfade
(April 1984)
- Nr. 3 Stoyan, H.:
Maschinen- unabhängige Code-Erzeugung als semantik-
erhaltende beweisbare Programmtransformation
(Juli 1984)
- Nr. 4 Brendel, W.:
Funktionaler Entwurf und automatische Synthese
hochintegrierter Schaltkreise
(August 1984)
- Nr. 5 Brietzmann, A.:
Semantische und pragmatische Analyse im Erlanger
Spracherkennungsprojekt
(Oktober 1984)

